O'REILLY®

Kubernetes на практике

Создание успешных платформ приложений





Джош Россо, Рич Ландер,
Александр Бранд, Джон Харрис

Production Kubernetes

Building Successful Application Platforms

Josh Rosso, Rieh Lander,
Alexander Brand, and John Harris

Beijing • Boston • Farnham • Sehastopol • Tokyo



Джош Россо, Рич Ландер,
Александр Бранд, Джон Харрис

Kubernetes на практике

Создание успешных платформ приложений

Санкт-Петербург
«БХВ-Петербург»
2022

УДК 004.273
ББК 32.973-018.2
Р77

Россо, Д.

Р77 Kubernetes на практике: Пер. с англ. / Д. Россо, Р. Ландер, А. Бранд, Д. Харрис. — СПб.: БХВ-Петербург, 2022. — 496 с.: ил.

ISBN 978-5-9775-1210-7

Книга посвящена практическому применению платформы Kubernetes. Подробно рассматривается архитектура Kubernetes и ее составные компоненты. Описаны модели развертывания инфраструктуры, ее топология, принципы автоматизации процессов, среда выполнения контейнеров, хранилища данных и сетевое взаимодействие между элементами системы. Рассматриваются создание и маршрутизация сервисов, управление конфиденциальными данными, допусками, мультитенантность, уровни изоляции и абстрагирование. Приведены наглядные примеры развертывания Kubernetes и оркестрации контейнеров для решения различных практических задач.

Для системных архитекторов, разработчиков ПО, специалистов по информационной безопасности

УДК 004.273
ББК 32.973-018.2

Научный редактор:

Архитектор решений, руководитель группы архитекторов и системных инженеров Croc Code — *Дмитрий Бардин*

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Людмила Гауль*
Перевод с английского	*Михаила Райтмана*
Компьютерная верстка	*Натальи Смирновой*
Оформление обложки	*Зои Канторович*

Подписано в печать 30 06.22
Формат 70×100$^1/_{16}$. Печать офсетная. Усл. печ. л. 39,99
Тираж 1200 экз Заказ № 4689.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-1-492-09230-8 (англ.)
ISBN 978-5-9775-1210-7 (рус.)

Оглавление

Предисловие

С тех пор как мы сделали платформу Kubernetes общедоступной, прошло уже больше шести лет. Я участвовал в данном проекте с самого начала и, между прочим, зафиксировал в его репозитории первую порцию кода (все выглядело не настолько впечатляюще, как может показаться; это был всего лишь формальный этап создания нового репозитория для публичного выпуска). Могу с уверенностью сказать, что успех Kubernetes оказался для нас довольно неожиданным; он стал возможным благодаря тесному сотрудничеству преданных и доброжелательных участников проекта, а также помощи тех, кто использует Kubernetes на практике.

Мне посчастливилось поработать с авторами этой книги в стартапе (Heptio), который был основан (в том числе и мною) для того, чтобы способствовать внедрению Kubernetes в типичные предприятия. Причиной успеха Heptio во многом стали усилия моих коллег по налаживанию прямой связи с теми пользователями Kubernetes, которые решают реальные проблемы. Я благодарен каждому из них. Данная книга вобрала в себя этот опыт, полученный из первых рук. Она предоставляет командам разработчиков инструменты, необходимые для успешного применения Kubernetes.

Моя профессиональная карьера с самого начала была ориентирована на построение систем, предназначенных для тех, кто разрабатывает приложения, как в одиночку, так и в составе команды. Все началось с Microsoft Internet Explorer и продолжилось работой над Windows Presentation Foundation, после чего я переключился на облачные технологии, такие как Google Compute Engine и Kubernetes. Снова и снова я наблюдал за тем, как создатели платформ сталкивались со своего рода проклятием. Те, кто занимается разработкой платформ, мыслят более долгосрочными категориями и сосредоточены на создании фундамента, который, как они надеются, прослужит не одно десятилетие. Но из-за этого они теряют из виду проблемы, с которыми пользователи сталкиваются прямо сейчас. Нередко мы настолько увлекаемся процессом разработки, что у нас не остается времени на оценку того, что мы создали.

Снять это проклятие можно только путем активного сбора информации за пределами тесного сообщества создателей платформ. Для меня это стало возможным благодаря работе в команде инженерно-технического обеспечения Heptio (и позже в группе проектирования Kubernetes [Kubernetes Architecture Team или KAT] в VMware). В результате удалось успешно внедрить Kubernetes разнообразным клиентам в ряде отраслей, а также выяснить, как на самом деле применяется "теория", заложенная в нашу платформу.

Данная проблема усугубляется еще и потому, что вокруг Kubernetes и CNCF (Cloud Native Computing Foundation) сформировалась активно развивающаяся "экосистема" из проектов, входящих в фонд CNCF, и тех, которые не имеют к нему прямого

отношения. Я люблю называть подобную экосистему "прекрасным хаосом". Это джунгли, в которых произрастают проекты разной степени зрелости, иногда с одинаковыми возможностями. Именно так выглядят инновации! Однако исследование экосистемы, как и джунглей, является рискованным и требует самоотверженности и настойчивости. Новичкам в мире Kubernetes зачастую не хватает времени или умений для того, чтобы как следует овладеть экосистемой этой платформы.

В данной книге описываются те части экосистемы, в которых уместно применять отдельные инструменты и средства, а также демонстрируется, как подобрать правильное решение частных проблем, с которыми сталкивается читатель. Но мы не просто советуем читателю выбрать тот или иной инструмент. Наш подход более общий, он требует понимания сути проблемы, решаемой с помощью определенной категории инструментов, уверенности в наличии этой конкретной проблемы, осведомленности о преимуществах и недостатках разных методов, и предлагает практические рекомендации относительно того, с чего начать. Это бесценная информация для тех, кто собирается применять Kubernetes в реальных условиях!

В завершение хотел бы сказать большое спасибо Джошу, Ричу, Алексу и Джону. Их опыт имеет прямое отношение к успеху многих клиентов. Они научили меня многому при работе над проектом, который мы основали более шести лет назад, и теперь множество других пользователей могут воспользоваться нашим уникальным опытом, прочитав эту книгу.

Джо Беда,
главный инженер VMware Tanzu,
соавтор Kubernetes,
Сиэтл, январь 2021 года.

Введение

Kubernetes — необычайно мощная технология, получившая стремительный взлет популярности. Она подготовила почву для реального прогресса в сфере управления процессом развертывания программного обеспечения. В момент появления Kubernetes ПО на основе API и распределенные системы, может, и не были широко распространены, но уже хорошо себя зарекомендовали. Проект Kubernetes стал отличным воплощением этих принципов, что послужило залогом его успеха. Но вместе с тем он привнес еще кое-что крайне важное. В недалеком прошлом ПО, способное самостоятельно достигать объявленного, желаемого состояния, существовало только в гигантских технологических компаниях, в которых работали группы самых талантливых инженеров. Теперь же высокодоступные, самовосстанавливающиеся, автоматически масштабирующиеся средства развертывания доступны любой организации, и все благодаря Kubernetes. Перед нами открывается будущее, в котором программные системы принимают от нас высокоуровневые команды общего характера и, действуя в соответствии с ними, обеспечивают нужные результаты за счет анализа сложившихся условий, преодоления постоянно меняющихся преград и решения проблем без нашего вмешательства. Более того, эти системы будут справляться со своими задачами быстрее и надежней, чем люди, работающие вручную. Благодаря Kubernetes мы с вами приблизились к этому будущему. Однако за мощь и богатые возможности приходится платить повышенной сложностью. Решение написать эту книгу было мотивировано нашим желанием поделиться своим опытом и помочь другим совладать со сложными аспектами данной платформы.

Если вы хотите использовать Kubernetes для создания приложений промышленного уровня, эта книга для вас. Если же вам нужно познакомиться с системой Kubernetes или понять, как она работает, то лучше поискать какую-то другую книгу. Эти темы раскрыты во множестве других изданий, в официальной документации, в бесчисленных статьях и в самом исходном коде. Советуем вам подкрепить чтение этой книги самостоятельным исследованием и тестированием обсуждаемых здесь решений, так как мы редко предоставляем примеры в виде подробного пошагового руководства. Мы пытаемся охватить столько теоретического материала, сколько необходимо, а процесс реализации в основном оставляем читателю в качестве упражнения.

На страницах этой книги вы найдете рекомендации о параметрах, инструментарии, шаблонах проектирования и их практическом применении. Читатель должен понимать, как авторы относятся к практике создания платформ приложений. Мы инженеры и архитекторы; мы сотрудничаем со многими компаниями из рейтинга Fortune 500, помогая им воплотить представления о том, как должна выглядеть платформа в производственном окружении. Kubernetes помогает нам в этом аж с

2015 года, когда вышла версия 1.0. Мы сделали все возможное для того, чтобы сосредоточиться на проектировании и философии, а не на инструментарии, поскольку новые инструменты появляются настолько быстро, что мы не успеваем о них писать. Тем не менее, мы обязаны продемонстрировать эти шаблоны проектирования с применением самых подходящих на сегодняшний день инструментов.

Нам удалось добиться значительных успехов, помогая группам разработчиков на их пути к переходу в облако и полной трансформации процессов создания и развертывания своего программного обеспечения. Но не обошлось и без неудач, частой причиной которых было непонимание руководства организации того, какие задачи решает Kubernetes. Вот почему мы с самого начала удаляем этому пристальное внимание. За это время мы обнаружили несколько областей, которые в особенности интересуют наших клиентов. Совещания, направленные на то, чтобы помочь нашим клиентам продвинуться на пути к развертыванию версий или даже просто определить этот путь, стали рутинными. Нам приходилось проводить их настолько часто, что мы решили написать книгу!

Мы снова и снова помогали организациям в прохождении этого пути к развертыванию их ПО, но во всех этих случаях просматривалась лишь одна закономерность: данный процесс каждый раз выглядит иначе, хотя иногда нам очень хотелось, чтобы это было не так. Мы хотим, чтобы вы отдавали себе отчет в том, что в настоящей книге не стоит искать "программы из пяти шагов" по развертыванию кода в производственном окружении или "10 вещей, которые должен знать каждый пользователь Kubernetes". Здесь мы будем обсуждать многочисленные моменты принятия решений и ловушки, с которыми мы сталкивались, при необходимости мы будем приводить конкретные примеры и случаи из жизни. Общепринятые рекомендации существуют, но к ним всегда следует относиться прагматично. Нет какого-то одного универсального подхода, а фраза "зависит от обстоятельств" является совершенно справедливым ответом на многочисленные вопросы, с которыми вы неизбежно столкнетесь на своем пути.

Тем не менее, мы призываем вас относиться к этой книге критически! Работая с клиентами, мы всегда приветствуем критику и предложения с их стороны. Знания необходимо постоянно оттачивать, поэтому мы периодически обновляем наши методики с учетом новых возможностей, информации и ограничений. Вы должны следовать нашему примеру. Поскольку сфера облачных технологий продолжает эволюционировать, вы, несомненно, будете отклоняться от наших рекомендаций. Мы лишь делимся здесь своим опытом, чтобы вы могли сопоставить нашу точку зрения со своей собственной.

Условные обозначения

В этой книге приняты следующие типографские обозначения:

♦ *Курсивный шрифт*

Служит для выделения новых терминов, URL-адресов, адресов электронной почты, имен и расширений файлов.

♦ `Моноширинный шрифт`

Предназначен для записи листингов программ, а также для выделения в тексте таких элементов, как имена переменных и функций, базы данных, типы данных, переменные окружения, операторы и ключевые слова.

♦ **`Полужирный моноширинный шрифт`**

Используется для выделения команд или другого текста, который пользователь должен вводить без каких-либо изменений.

♦ *`Курсивный моноширинный шрифт`*

Обозначает те элементы в коде, которые требуется заменить предоставленными пользователем значениями или значениями, зависящими от контекста.

Термины, относящиеся к Kubernetes, начинаются с большой буквы: Pod-оболочка[*], Сервис и StatefulSet.

Чтобы акцентировать внимание читателя, в некоторых местах в текст вставлены соответствующие значки:



Данный элемент обозначает совет или предложение.

Этот значок обозначает примечание общего характера.

Данный элемент обозначает предупреждение или предостережение.

Использование примеров кода

Дополнительный материал (примеры программного кода, упражнения и т. д.) доступен для загрузки по адресу **https://github.com/production-kubernetes**.

Если при использовании примеров кода у вас возникнут технические вопросы или проблемы, пожалуйста, обращайтесь к нам по адресу **bookquestions@oreilly.com**.

Цель этой книги — помочь в решении ваших задач. Если предлагается какой-то пример кода, его разрешается использовать в ваших программах и документации. Обращаться к нам, если вы не воспроизводите его существенную часть, не нужно, например, в ситуации, когда вы включаете в свою программу несколько фрагментов кода, приведенного здесь. Однако продажа или распространение CD-ROM с примерами из книг издательства O'Reilly требует отдельного разрешения. Цитировать эту книгу с примерами кода при ответе на вопрос вы можете свободно, но, если хотите включить существенную часть приведенного здесь кода в документацию своего продукта, вам следует связаться с нами.

[*] В тексте сохранен оригинальный английский термин Pod. — *Прим. ред.*

Мы приветствуем, но не требуем наличия ссылки на оригинал. Ссылка обычно состоит из названия, имени автора, издательства, ISBN и копирайта. Например, *Kubernetes на практике*, Джош Россо, Рич Ландер, Александр Бранд и Джон Харрис (O'Reilly). Copyright 2021 Josh Rosso, Rich Lander, Alexander Brand, and John Harris, 978-1-492-09231-5.

Если вы сомневаетесь в том, что работа с примерами кода не выходит за рамки добросовестного использования или не нарушает условий, перечисленных выше, то можете обратиться к нам по адресу **permissions@oreilly.com**.

Платформа онлайн-обучения O'Reilly

На протяжении более 40 лет издательство O'Reilly Media (**http://oreilly.com**) проводит технические и бизнес-тренинги, делится знаниями и опытом, чтобы помочь компаниям достичь успеха.

Наше уникальное сообщество экспертов и новаторов делится знаниями и опытом с помощью книг, статей, конференций и нашей платформы онлайн-обучения. Эта платформа предоставляет доступ к учебным курсам, проводимым в прямом эфире, углубленные учебные программы, интерактивные среды для написания кода и обширную коллекцию текстов и видео от O'Reilly и более чем 200 других издательств. Больше информации можно найти на странице **http://oreilly.com**.

Благодарности

Авторы хотели бы поблагодарить Кэти Гаманжи, Майкла Гуднесса, Джима Уебера, Джеда Салазара, Тони Скалли, Монику Родригез, Криса Докери, Ральфа Бэнкстона, Стова Слоку, Аарона Миллера, Тунде Олу-Ису, Алекса Уитроу, Скотта Лоу, Райана Чаппла и Кенана Дервисевица за их отзывы о рукописи. Спасибо Полу Лундину за поддержку в написании этой книги и за создание замечательной команды инженерно-технического обеспечения в Heptio. Все члены этой команды внесли свой вклад, участвуя в развитии множества идей и практических подходов, которые рассматриваются на последующих страницах данной книги. Спасибо Джо Беде, Скотту Бьюкенену, Даниелле Барроу и Тиму Ковентри-Коксу из VMware за их поддержку в основании этого проекта и работе над ним. Наконец, мы благодарим Джона Девинса, Джеффа Блейела и Кристофера Фаучера из O'Reilly за их постоянную поддержку.

Авторы также хотели бы лично выразить признательность следующим людям:

Джош. Хотел бы поблагодарить Джессику Аппельбаум за ее невероятную поддержку, в частности за оладьи с голубикой, которые она мне готовила, пока я работал над этой книгой. Также хочу сказать спасибо моей маме, Анжеле, и папе, Джо, за то, что с детства были моей опорой.

Рич. Хотел бы поблагодарить мою жену, Тейлор, и детей, Райну, Жасмин, Макса и Джона, за их поддержку и понимание, которые они проявили, пока я работал над

этой книгой. Также хочу сказать спасибо моей маме, Дженни, и папе, Норму, за то, что были отличными примерами для подражания.

Александр. Шлю свои любовь и благодарность моей жене, Анаис, которая оказывала мне всестороннюю поддержку, пока я уделял время написанию этой книги. Также признателен моей семье, друзьям и коллегам, которые помогли мне стать тем, кем я являюсь сегодня.

Джон. Хотел бы поблагодарить мою прекрасную жену, Кристину, за ее любовь и терпение во время моей работы над этой книгой. Также говорю спасибо моим близким друзьям и семье за непрерывные помощь и поддержку на протяжении всех этих лет.

Издательство «БХВ» искренне благодарит за помощь в подготовке русскоязычного издания этой книги научного редактора и рецензента — руководителя группы архитекторов и системных инженеров Croc Code Дмитрия Бардина.

ГЛАВА 1

Путь к эксплуатации

За годы своего существования платформу Kubernetes внедрили многие организации. Росту ее популярности, бесспорно, способствует распространение контейнерных рабочих заданий и наличие микросервисов. Когда команды, занимающиеся эксплуатацией, обслуживанием инфраструктуры и разработкой, сталкиваются с необходимостью создания, выполнения и поддержки этих рабочих процессов, некоторые из них выбирают в качестве решения Kubernetes. Это довольно молодой проект в сравнении с другими продуктами с открытым исходным кодом, такими как Linux. Как могут подтвердить многие клиенты, с которыми мы сотрудничали, большинство пользователей Kubernetes только начинают работать с этой платформой. Несмотря на ее применение во многих организациях, она редко встречается в эксплуатации и еще реже — в высоконагруженных проектах. В этой главе мы подготовимся к пути, на котором находятся многие команды инженеров, работающие с Kubernetes. В частности, мы рассмотрим некоторые аспекты, на которые следует обращать внимание при планировании процесса подготовки проекта к эксплуатации.

Что такое Kubernetes

Что такое Kubernetes? Платформа? Инфраструктура? Приложение? На этот счет есть много мнений, приверженцы которых могут дать вам свое определение. Вместо того чтобы высказать еще одно мнение, мы сосредоточимся на классификации тех задач, которые решает Kubernetes. После этого мы поговорим о том, как, опираясь на этот набор возможностей, достигнуть результатов, соответствующих производственному уровню. Идеальный исход прочтения этой книги — ситуация, при которой рабочие задания успешно работают с реальным трафиком.

Название *Kubernetes* в какой-то степени носит общий характер. Если заглянуть на GitHub, можно заметить, что организация kubernetes содержит (на момент написания этих строк) 69 репозиториев. А ведь есть еще и kubernetes-sigs со 107 репозиториями. Не говоря уже о сотнях проектов в составе фонда CNCF (Cloud Native Computing Foundation), существующих в этой среде! В данной книге под Kubernetes имеется в виду лишь основной проект с одноименным названием. Какой из них основной? Тот, который находится в репозитории kubernetes/kubernetes (**https://github.com/kubernetes/kubernetes**). Именно там хранятся все компоненты, которые встречаются в большинстве кластеров. От кластера, состоящего из этих компонентов, можно ожидать следующих возможностей:

- планирование распределения рабочей нагрузки для множества хостов;
- предоставление декларативного, расширяемого API-интерфейса для взаимодействия с системой;

- наличие утилиты командной строки `kubectl` для взаимодействия с API-сервером в ручном режиме;
- приведение объектов от текущего состояния к желаемому;
- предоставление базового служебного слоя для направления запросов к приложениям и обратно;
- наличие нескольких интерфейсов для поддержки подключаемых модулей, отвечающих за работу с сетью, хранилищем и т. д.

Перечисленные возможности делают данный проект, согласно утверждению его разработчиков, *системой оркестрации контейнеров, готовой к эксплуатации*. Говоря простым языком, Kubernetes дает нам возможность исполнять и распределять контейниризированные приложения на разных хостах. Помните об этом по мере того, как мы будем углубляться в подробности. Надеемся, со временем нам удастся продемонстрировать, что эта возможность, несмотря на ее основополагающий характер, является лишь одним из этапов на пути к эксплуатации.

Основные компоненты

Какие компоненты обеспечивают функциональность, которая рассматривается в этой книге? Как уже упоминалось, основные компоненты находятся в репозитории kubernetes/kubernetes. Многие из нас применяют их по-разному. Например, те, кто пользуются управляемыми сервисами наподобие Google Kubernetes Engine (GKE), скорее всего, уже имеют все эти компоненты на своих хостах. Кто-то может скачивать исходный код из репозиториев или получать подписанные версии от поставщика. Как бы то ни было, выпуски Kubernetes доступны для свободного скачивания в репозитории kubernetes/kubernetes. Распаковав такой выпуск, вы можете получить соответствующие двоичные файлы с помощью команды `cluster/get-kubebinaries.sh`. Она автоматически определит вашу целевую архитектуру и скачает серверные и клиентские компоненты. Давайте рассмотрим это на примере следующего кода и затем проанализируем ключевые компоненты:

```
$ ./cluster/get-kube-binaries.sh

Kubernetes release: v1.18.6
Server: linux/amd64  (to override, set KUBERNETES_SERVER_ARCH)
Client: linux/amd64  (autodetected)

Will download kubernetes-server-linux-amd64.tar.gz from https://dl.k8s.io/v1.18.6
Will download and extract kubernetes-client-linux-amd64.tar.gz
Is this ok? [Y]/n
```

Внутри архива со скачанными серверными компонентами, который, скорее всего, был сохранен в папку server/kubernetes-server-${ARCH}.tar.gz, находятся ключевые элементы, составляющие кластер Kubernetes:

- *API-сервер* — главное место взаимодействия для всех компонентов Kubernetes и пользователей. Именно здесь мы получаем, добавляем, удаляем и изменяем объекты. API-сервер делегирует работу с состоянием внутреннему компоненту, роль которого чаще всего играет etcd.

- *Kubelet* — агент, размещенный на хосте и взаимодействующий с API-сервером для получения информации о состоянии узла и определения того, выполнение каких приложений следует на нем планировать. В пределах хоста он взаимодействует со средой выполнения контейнеров, такой как Docker, обеспечивая корректные запуск и выполнение приложений, запланированных на соответствующем узле.
- *Диспетчер контроллеров* — группа обработчиков, которая объединена в одно приложение и занимается приведением многих важных объектов Kubernetes к желаемому состоянию. Когда такое состояние (например, "в Развертывании должно быть три реплики") объявляется, для его достижения внутренний контроллер берет на себя создание новых подов.
- *Планировщик* определяет, где должны выполняться приложения, в зависимости от того, какой узел он считает оптимальным. Для принятия этого решения он использует фильтрацию и систему оценок.
- *Kube-proxy* реализует сервисы Kubernetes, предоставляя виртуальные IP-адреса, способные направлять трафик к подам. Это достигается за счет механизма фильтрации пакетов на хосте, такого как `iptables` или `ipvs`.

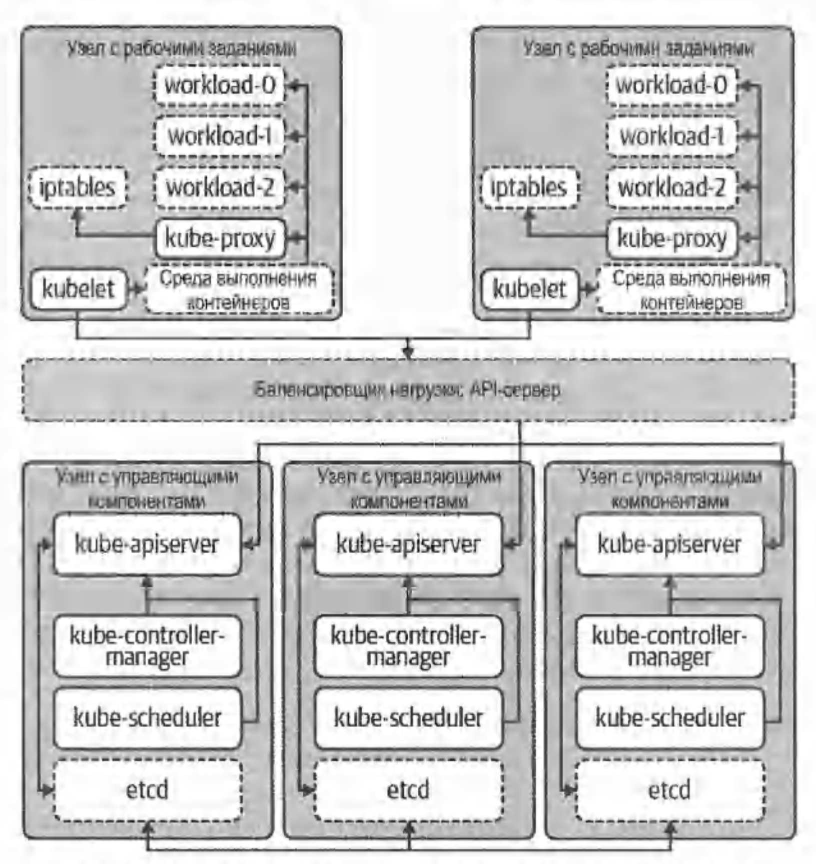


Рис. 1.1. Ключевые компоненты, составляющие кластер Kubernetes. Компоненты, обрамленные пунктирной линией, не входят в ядро Kubernetes

Несмотря на то, что это не исчерпывающий список, но в нем перечислены главные компоненты, составляющие основу той функциональности, которую мы обсудили. Если говорить в контексте архитектуры, то на рис. 1.1 показано, как взаимодействуют эти компоненты.



Архитектура Kubernetes имеет много вариантов. Например, во многих кластерах компоненты kube-apiserver, kube-scheduler и kube-controller-manager выполняются в виде контейнеров. Это означает, что мастер-узел или управляющий слой (control-plane) может также включать container-runtime, kubelet и kube-proxy. Подобные аспекты развертывания будут рассмотрены в следующей главе.

Не только оркестрация: дополнительные функции

В некоторых областях Kubernetes не ограничивается одной лишь оркестрацией приложений. Как уже упоминалось, компонент kube-proxy делает так, чтобы хосты автоматически предоставляли приложениям виртуальные IP-адреса, которые ведут к одной или нескольким внутренним подам. Очевидно, что это выходит за рамки выполнения и планирования контейнеризированных приложений. Вместо реализации всего этого в ядре Kubernetes теоретически можно было бы определить API-интерфейс для работы с сервисами, требуя, чтобы он был реализован в виде подключаемого модуля. В результате такого подхода мы были бы вынуждены выбирать между многочисленными модулями, доступными в экосистеме, вместо того чтобы задействовать функциональность, встроенную в ядро.

Это модель, которая применяется во многих API-интерфейсах Kubernetes, таких, как NetworkPolicy. Например, создание объекта `Ingress` в кластере Kubernetes не гарантирует выполнение действия. Иными словами, API-интерфейс хоть и существует, но находится за пределами ядра. Разработчики должны выбрать технологию, на основе которой этот API-интерфейс будет реализован. В случае с `Ingress` многие используют такой контроллер, как ingress-nginx (**https://kubernetes.github.io/ingress-nginx**), который работает в кластере. Он реализует API-интерфейс, считывая объекты `Ingress` и создавая конфигурационные файлы для экземпляров NGINX, которые указывают поды. Однако ingress-nginx — это лишь один из множества вариантов. Project Contour (**https://projectcontour.io**) реализует тот же Ingress API, но при этом конфигурирует экземпляры Envoy, прокси-сервера, лежащего в основе Contour. Благодаря наличию подключаемых модулей разработчикам есть из чего выбрать.

Интерфейсы Kubernetes

Развивая эту идею расширения функциональности, мы должны теперь рассмотреть интерфейсы. Интерфейсы Kubernetes позволяют нам изменять и дополнять основные возможности этой платформы. Мы относимся к интерфейсу как к определению или контракту о том, как должно происходить взаимодействие с тем или иным компонентом. В сфере разработки программного обеспечения это созвучно принципу определения функциональности, которую могут реализовывать классы или структуры. В таких системах, как Kubernetes, для удовлетворения требований этих

интерфейсов мы развертываем подключаемые модули, предоставляя возможности наподобие управления сетью.

Конкретным примером отношений между интерфейсами и подключаемыми модулями является CRI (Container Runtime Interface — интерфейс среды выполнения контейнеров; **https://github.com/kubernetes/cri-api**). На ранних этапах развития платформа Kubernetes поддерживала всего одну среду выполнения контейнеров, Docker. И хотя Docker по-прежнему присутствует во многих кластерах, наблюдается растущий интерес к альтернативам, таким как containerd (**https://containerd.io**) и CRI-O (**https://github.com/cri-o/cri-o**). На рис. 1.2 вышеупомянутые отношения проиллюстрированы на примере этих двух сред выполнения контейнеров.

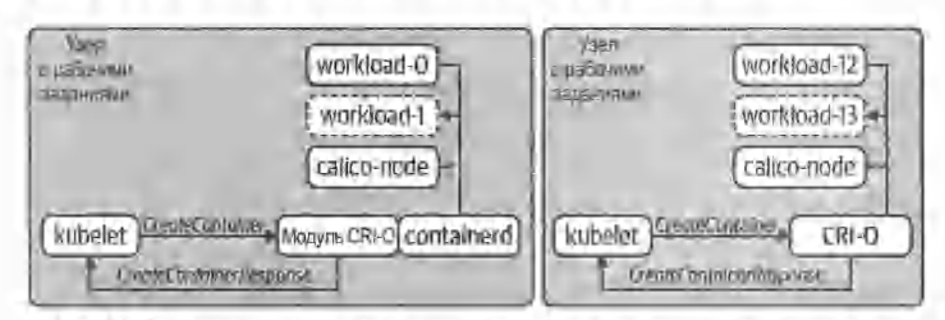


Рис. 1.2. Две разные среды выполнения контейнеров, установленных на двух узлах. Kubelet шлет такие запросы, как `CreateContainer`, определенные в CRI, в расчете на то, что среда выполнения их удовлетворит и вернет ответ

Во многих интерфейсах такие команды, как `CreateContainerRequest` или `PortForwardRequest` выполняются в виде удаленного вызова процедур (англ. Remote Procedure Calls или RPC). В случае с CRI взаимодействие происходит по gRPC, а kubelet ожидает получения ответов вроде `CreateContainerResponse` и `PortForwardResponse`. На рис. 1.2 можно также увидеть две разные модели удовлетворения требований CRI. Модуль CRI-O с самого начала создавался в качестве реализации CRI, поэтому kubelet передает ему эти команды напрямую. А вот containerd поддерживает подключаемый модуль, который служит прослойкой между kubelet и собственными интерфейсами. Какой бы ни была конкретная архитектура, главное, чтобы агенту kubelet не нужно было знать о том, как организована работа среды выполнения контейнеров в *каждом* отдельном случае. Именно этот принцип делает интерфейсы настолько мощным средством проектирования, разработки и развертывания кластеров Kubernetes.

Мы видели, как со временем часть функций была вынесена из основного проекта в соответствии с упомянутым принципом подключаемых модулей. Речь идет о вещах, которые существовали в кодовой базе kubernetes/kubernetes по историческим причинам. Примером этого являются средства интеграции cloud-provider (**https://github.com/kubernetes/cloud-provider**; CPI). Большинство модулей CPI традиционно встраивались в такие компоненты, как kube-controller-manager и kubelet. Они, как правило, отвечали за выделение балансировщиков нагрузки, предоставление доступа к метаданным облачных провайдеров и пр. Иногда, особенно

до создания CSI (Container Storage Interface — интерфейс хранилищ для контейнеров; **https://kubernetes-csi.github.io/docs/introduction.html**), эти провайдеры выделяли блочные хранилища и делали их доступными для приложений, выполняемых в Kubernetes. Это довольно обширная функциональность даже для ядра, не говоря уже о том, что ее нужно было заново реализовывать для каждого отдельно взятого провайдера! В качестве более разумного решения эти возможности были вынесены в отдельную модель интерфейса, kubernetes/cloud-provider (**https://github.com/kubernetes/cloud-provider**), реализацией которой могут заниматься разные проекты и поставщики. В результате удалось не только упорядочить кодовую базу Kubernetes, но и обеспечить возможность управления этой функциональностью вне основных кластеров Kubernetes. Сказанное относится и к таким распространенным процедурам, как обновление и устранение уязвимостей.

На сегодня существует несколько интерфейсов, которые позволяют видоизменять и дополнять возможности Kubernetes. Вот общий список, который мы расширим в следующих главах этой книги:

- CNI (Container Networking Interface — интерфейс управления сетью контейнеров) позволяет сетевым провайдерам определять, как они выполняют свои функции, от IPAM до непосредственной маршрутизации пакетов.
- CSI (Container Storage Interface — интерфейс хранилищ для контейнеров) позволяет провайдерам хранилищ удовлетворять запросы рабочих заданий в пределах кластера. Обычно реализуется для таких технологий, как vSAN и EBS.
- CRI (Container Runtime Interface — интерфейс среды выполнения контейнеров) обеспечивает поддержку различных сред выполнения, в том числе таких популярных, как Docker, containerd и CRI-O. Также способствовал распространению менее традиционных сред выполнения, включая firecracker, которая использует KVM для создания минимальных виртуальных машин.
- SMI (Service Mesh Interface — интерфейс mesh-сети сервисов) — это один из новых интерфейсов, появившихся в экосистеме Kubernetes. Он создавался в надежде унифицировать определение таких понятий, как "политика маршрутизации трафика", "телеметрия" и "управление".
- CPI (Cloud Provider Interface — интерфейс облачных провайдеров) позволяет провайдерам, таким как VMware, AWS, Azure и др., создавать для своих облачных сервисов точки интеграции с кластерами Kubernetes.
- Спецификация среды выполнения OCI (Open Container Initiative) стандартизирует форматы образов так, чтобы образы контейнеров, созданные в соответствии с ней, могли выполняться в любой среде, совместимой с OCI. Она не имеет прямого отношения к Kubernetes, но поддерживает стремление к реализации подключаемых сред выполнения контейнеров (CRI).

Kubernetes в целом

Теперь давайте поговорим о сфере применения Kubernetes. Это *средство оркестрации контейнеров* с некоторыми дополнительными функциями. Его можно рас-

ширять и изменять за счет подключаемых модулей для интерфейсов. Kubernetes может стать основополагающим инструментом для организаций, которым нужен изящный способ выполнения своих приложений. Но давайте сделаем шаг назад. Если взять системы для выполнения приложений, которые используются в вашей организации в настоящий момент, и заменить их платформой Kubernetes, будет ли этого достаточно? Во многих случаях компоненты и механизмы, составляющие "платформу приложений", имеют множество нюансов.

Нам неоднократно приходилось наблюдать за муками, вызванными необдуманным выбором так называемой "стратегии Kubernetes", когда руководство организации верит в то, что Kubernetes послужит толчком к модернизации процессов разработки и функционирования ПО. Kubernetes — это замечательная технология, но она не должна определять направление развития вашей организации в отношении инфраструктуры, платформы и/или программного обеспечения. Мы извиняемся, если вам это и так понятно, но вы бы удивились, узнав, сколько исполнительных специалистов и высокопоставленных руководителей, с которыми мы разговаривали, считают, что платформа Kubernetes сама по себе является решением проблем, в то время как на самом деле их проблемы были связаны с доставкой приложений, разработкой программного обеспечения или организационными/кадровыми вопросами. К Kubernetes лучше относиться как к одному из элементов инфраструктуры, который позволяет создавать инфраструктурную платформу для ваших приложений. Мы все ходим вокруг да около идеи о платформе приложений, поэтому давайте поговорим о том, что это такое.

Что такое платформа приложений

Одним из ключевых этапов на нашем пути к эксплуатации должно стать обсуждение идеи о платформе приложений. В нашем понимании *платформа приложений* — это подходящее место для выполнения приложений. Как и с большинством определений в данной книге, его практическое воплощение зависит от конкретной организации. Разные структурные подразделения организации ориентируются на такие желаемые результаты, как удовлетворенность разработчиков, снижение эксплуатационных расходов, сокращение времени цикла обратной связи при доставке ПО и т. д. Платформа приложений зачастую находится на пересечении приложений и инфраструктуры. Удобство разработки (англ. developer experience или devx) обычно является ключевым принципом в этой области.

Платформы приложений бывают разными. Некоторые в основном инкапсулируют такие основополагающие компоненты, как IaaS (например, AWS) или средство оркестрации (например, Kubernetes). Отличным примером этой модели является Heroku. С помощью данного сервиса вы можете взять проект, написанный на таких языках, как Java, PHP или Go, и развернуть его в реальных условиях эксплуатации с помощью одной команды. Ваше приложение работает в окружении множества сервисов, которые пришлось администрировать самостоятельно, если бы они не предоставлялись самой платформой. Имеются в виду такие вещи, как средства сбора метрик, сервисы для работы с данными и непрерывная доставка (англ. Continuous

Delivery или CD). Платформа также дает вам базовые конфигурации для запуска набора наиболее распространенных приложений, которые могут легко масштабироваться. Используется ли Kubernetes в Heroku? Есть ли у Heroku собственные центры обработки данных (ЦОД), или все работает поверх AWS? Какая разница? Для пользователей Heroku эти детали неважны. Важно то, что эти задачи делегируются провайдеру или платформе, что позволяет разработчикам больше времени уделять решению бизнес-проблем. Такой подход применяется не только в облачных сервисах. В OpenShift от RedHat принята похожая модель, в которой Kubernetes является, скорее, одной из деталей реализации, а разработчики и администраторы платформы взаимодействуют с набором более высокоуровневых абстракций.

Почему бы на этом не остановиться? Если такие платформы, как Cloud Foundry, OpenShift и Heroku уже решили эти проблемы за нас, зачем возиться с Kubernetes? Существенным недостатком многих готовых платформ приложений является то, что они навязывают нам свои "взгляды". Делегирование всех обязанностей по управлению и обслуживанию внутренней части системы позволяет избежать большого объема работ по сопровождению. Но в то же время, если методы обнаружения сервисов или управления конфиденциальными данными, которые применяет платформа, не соответствуют вашим организационным требованиям, вы можете быть лишены возможности решить эту проблему. Кроме того, есть такое понятие как зависимость от поставщика или от чужого мнения. Абстракции влекут за собой определенные взгляды на то, как должны проектироваться, упаковываться и развертываться ваши приложения. Это означает, что переход на другую систему может оказаться непростым. Например, рабочие задания намного проще перенести между Google Kubernetes Engine (GKE) и Amazon Elastic Kubernetes Engine (EKS), чем между EKS и Cloud Foundry.

Спектр подходов

Как вы уже сами видите, существует несколько подходов к созданию успешной платформы приложений. Давайте сделаем для демонстрации несколько далеко идущих предположений и сравним теоретические компромиссы, свойственные разным подходам. На рис. 1.3 за пример взята типичная компания среднего/крупного размера, с которой мы обычно имеем дело.

В левой нижней части мы видим развертывание самих кластеров Kubernetes, что требует относительно небольших усилий со стороны инженеров, особенно если за обслуживание управляющего уровня отвечает сторонний сервис вроде EKS. Такой подход обеспечивает не слишком высокую готовность к промышленному внедрению, так как большинство организаций сталкивается с необходимостью в дополнительной работе поверх Kubernetes. Но в некоторых ситуациях одной лишь платформы Kubernetes может быть достаточно, например, при использовании отдельных кластеров.

В правой нижней части находятся более зрелые платформы, которые предоставляют разработчикам полный набор готовых возможностей. Cloud Foundry — отличный пример проекта, который решает множество задач, возлагаемых на платформу

приложений. Программное обеспечение, работающее в Cloud Foundry, не должно выходить за те рамки, которые очертили разработчики этой платформы. А вот платформа OpenShift, которая куда лучше готова к промышленному использованию, чем просто Kubernetes, дает бóльшую свободу выбора относительно того, как ее настраивать. Что собой являет эта гибкость: преимущество или лишнюю работу? Это ключевой вопрос, на который вы должны ответить.

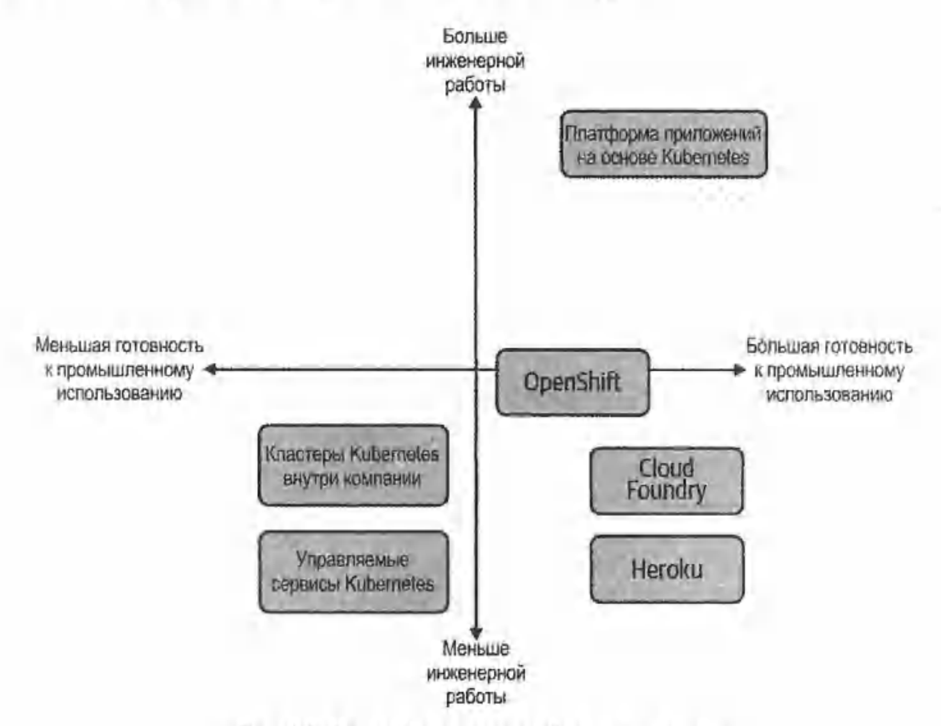


Рис. 1.3. Разнообразные способы организации платформы приложений для разработчиков

Наконец, справа вверху представлено создание платформы приложений поверх Kubernetes. Это, несомненно, требует больше всего инженерных усилий по сравнению с остальными вариантами, по крайней мере, с точки зрения платформы. Но благодаря расширяемости Kubernetes вы можете получить результат, который будет лучше соответствовать вашим потребностям в плане разработки, инфраструктуры и бизнеса.

Спектр подходов с учетом потребностей вашей организации

Диаграмме на рис. 1.3 недостает третьего измерения, оси *Z*, которая показывает, насколько тот или иной подход соответствует вашим требованиям. Рассмотрим еще одно графическое представление. На рис. 1.4 показано, как это может выглядеть, если учитывать соответствие платформы потребностям организации.

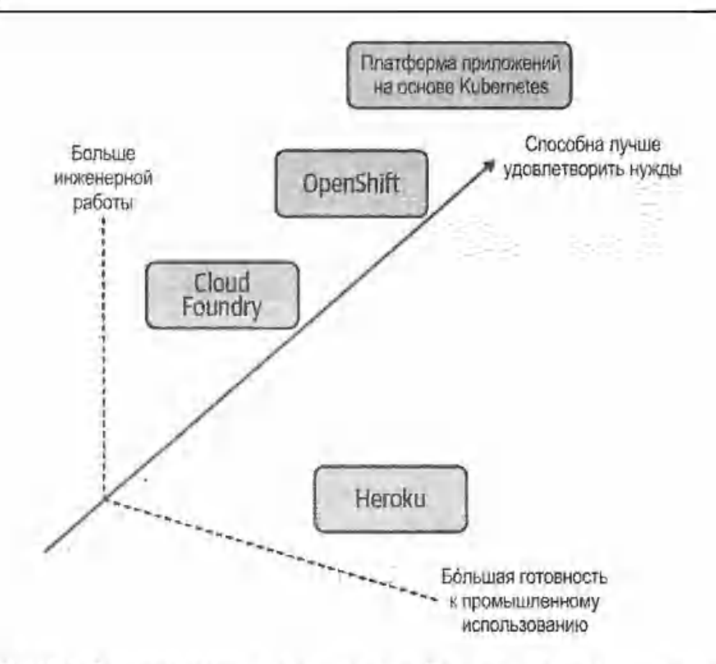


Рис. 1.4. Дополнительное измерение (ось *Z*), сопоставляющее варианты с потребностями вашей организации

С точки зрения требований, возможностей и ожидаемого поведения, самостоятельное создание платформы почти всегда дает наиболее подходящий результат. Или, по крайней мере, наиболее близкий к таковому. Ведь вы можете создать что угодно! Теоретически вы могли бы воссоздать в своей компании Heroku на основе Kubernetes, с небольшими изменениями функциональности, но при этом следует учитывать соотношение затрат и полученной выгоды (оси *X* и *Y*). Давайте добавим в наш разговор больше конкретики и рассмотрим некоторые требования к платформе приложений следующего поколения:

- регламент обязывает размещать основную часть системы локально;
- вам нужно поддерживать свои многочисленные физические серверы вместе с ЦОД, в котором используется vSphere;
- вы хотите удовлетворить растущий спрос со стороны разработчиков на упаковывание приложений в контейнеры;
- вам нужно создать API-интерфейс для механизмов самообслуживания, которые позволят отойти от выделения инфраструктурных ресурсов "на основе тикетов";
- вы хотите убедиться в том, что разрабатываемые вами API-интерфейсы не завязаны на конкретного поставщика, так как в прошлом миграция с такого рода систем стоила вам миллионы;
- вы не против платить за корпоративную поддержку разнообразных продуктов в своем стеке, но не в восторге от моделей, в которых весь стек лицензируется для каждого узла, ядра или экземпляра приложения.

Чтобы определить, является ли построение платформы приложений разумным начинанием, мы должны понимать, насколько "созрели" наши инженерные возможности, есть ли у нас желание заниматься формированием и развитием групп разработчиков, и обладаем ли мы достаточными ресурсами.

Платформы приложений: подводим итоги

Следует признать, что определение платформы приложений остается довольно размытым. Мы сосредоточились на разнообразных платформах, которые, как нам кажется, дают командам разработчиков нечто большее, чем просто оркестрацию приложений. Мы также ясно дали понять, что Kubernetes можно изменять и дополнять для получения похожих результатов. Выводя свое мышление за рамки того, как внедрить Kubernetes, и задаваясь вопросами наподобие "как сейчас организован рабочий процесс разработчиков, какие у них проблемы и желания?", группы сотрудников, которые занимаются платформой и инфраструктурой, будут более успешными в своей профессиональной деятельности. Можно утверждать, что, сосредоточившись на последнем, вы с большей вероятностью наметите себе подходящий путь к эксплуатации и справитесь с непростым процессом внедрения. В конечном счете мы хотим удовлетворить требования к инфраструктуре, безопасности и удобству разработки, чтобы наши клиенты (которыми обычно являются разработчики) получили решение, отвечающее их потребностям. Как правило, мы не стремимся предоставить "мощный" движок, поверх которого каждый разработчик должен создать собственную платформу, что с юмором продемонстрировано на рис. 1.5.



Рис. 1.5. Если разработчики хотят иметь готовое решение (например, полноценный автомобиль), не надейтесь, что им будет достаточно двигателя без кузова, колес и прочего

Создание платформ приложений на основе Kubernetes

Итак, мы воспринимаем Kubernetes как один из этапов нашего пути к развертыванию. Учитывая это, справедливым будет следующий вопрос: "Может, у Kubernetes просто недостаточно возможностей?" Проект Kubernetes был вдохновлен одним из важнейших принципов философии Unix: "пишите программы, которые делают что-то одно и делают это хорошо". Мы считаем, что лучшими возможностями этой платформы являются те, которых у нее нет! Это становится еще очевидней после

мучений с универсальными платформами, которые пытаются решать за вас все проблемы человечества. Блестящая черта платформы Kubernetes — ее узкая направленность: она пытается быть хорошим средством оркестрации, но в то же время предоставляет четкие интерфейсы, поверх которых можно написать что-то свое. Таким образом, платформу можно сравнить с фундаментом дома.

Хороший фундамент должен быть прочным, подходить для возведения остальных конструкций и предоставлять подходящие "интерфейсы" для проведения коммуникаций. Несмотря на важную роль, сам по себе фундамент редко когда становится подходящим местом для проживания. Обычно на фундаменте должен находиться какой-то дом. Прежде чем обсуждать *строительство* поверх такого фундамента, как Kubernetes, давайте рассмотрим готовое меблированное жилище, показанное на рис. 1.6.

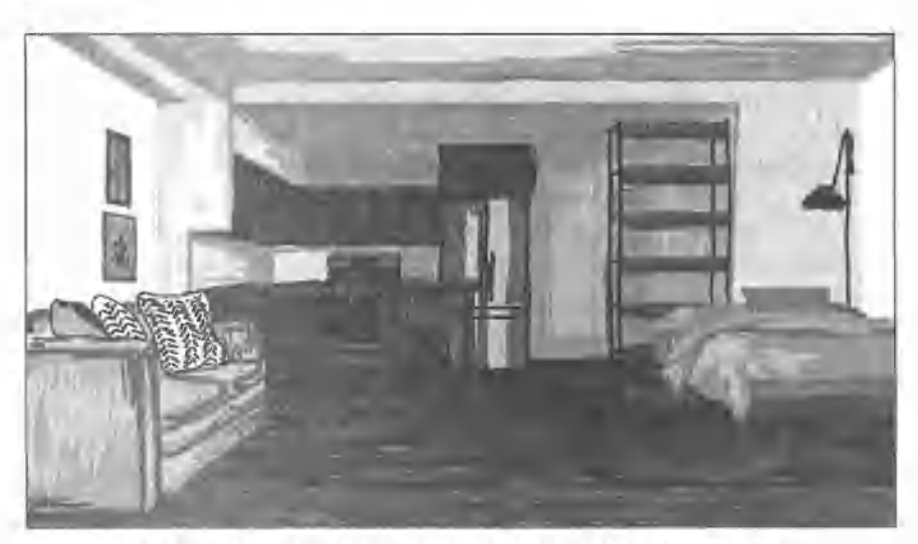


Рис. 1.6 Апартаменты, готовые к заселению. Похожи на PaaS, такие как Heroku (иллюстрация Джессики Аппельбаум)

Этот вариант жилья, подобный нашим примерам с Heroku, подходит для проживания и не требует дополнительных вложений. Вы, конечно, можете немного изменить интерьер, но большинство проблем уже решено. Если подходит арендная плата, и вы согласны смириться с тем, как все устроено внутри, вы можете жить припеваючи с самого первого дня.

Но вернемся к платформе Kubernetes, которую мы сравнили с фундаментом. Поверх нее можно построить жилой дом, как показано на рис. 1.7.

Инвестируя в планирование, проектирование и поддержку, мы можем создавать замечательные платформы, способные выполнять рабочие задания в разных организациях. Это означает, что мы контролируем каждый аспект конечного результата. Дом может и должен быть приспособлен к нуждам его будущих жильцов (наших приложений). Давайте теперь разберем различные слои и соображения, благодаря которым это возможно.

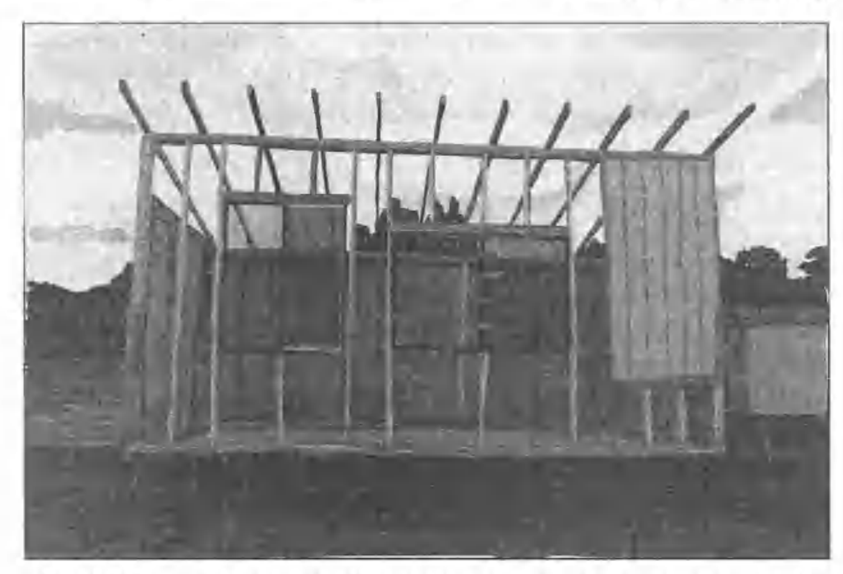


Рис. 1.7. Строительство дома. Похоже на создание платформы приложений, в основе которой лежит Kubernetes (иллюстрация Джессики Аппельбаум)

Начиная снизу

Мы должны начать с самого низа, в том числе с того, какие технологии будет использовать Kubernetes. Обычно речь идет о центре обработки данных или облачном провайдере, который предлагает вычислительные ресурсы, хранилище и доступ к сети. Дальше на основе этого можно развертывать Kubernetes. Всего за несколько минут вы можете получить кластер, который работает поверх внутренней инфраструктуры. Kubernetes можно развернуть несколькими способами, которые мы подробно обсудим в *главе 2*.

Следующий важный аспект существования кластеров Kubernetes, который поможет нам определить, что следует создавать поверх них, — процесс внедрения Kubernetes. Его ключевые элементы представлены на рис. 1.8.

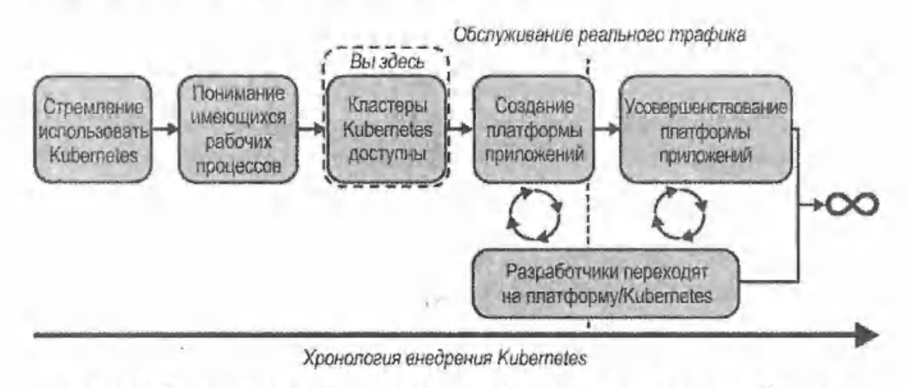


Рис. 1.8. Процесс, через который могут пройти наши команды разработчиков на пути к полноценному внедрению Kubernetes

С появлением Kubernetes в вашей организации, не удивляйтесь, если к вам начнут обращаться с подобными вопросами:

- "Как сделать так, чтобы трафик между приложениями шифровался?"
- "Как сделать так, чтобы исходящий трафик проходил через шлюз, который гарантирует CIDR?"
- "Как организовать трассировку и дашборды для приложений?"
- "Как проводить погружение разработчиков, не требуя от них глубоких знаний Kubernetes?"

Этот список можно продолжать бесконечно. Зачастую нам самим приходится определять, какие требования должны удовлетворяться на уровне платформы, а какие — на уровне приложения. Главное здесь — иметь глубокое понимание существующих рабочих процессов, чтобы создаваемая нами платформа отвечала текущим ожиданиям. Если мы не в состоянии обеспечить нужную функциональность, как это повлияет на группы разработчиков?

Дальше мы можем приниматься за создание платформы поверх Kubernetes. Очень важно, чтобы в ходе данного процесса мы действовали совместно с разработчиками, желающими опробовать нашу платформу на ранних этапах, и знали об их впечатлениях, поскольку это поможет нам принимать обоснованные решения с учетом оперативно поступающих отзывов. Такой процесс не должен прекращаться даже после развертывания платформы в промышленном окружении. Не рассчитывайте на то, что платформа получится статической, и что разработчики будут пользоваться ею на протяжении десятилетий. Чтобы достичь успеха, мы должны постоянно взаимодействовать с нашими группами разработчиков, что позволит нам узнавать о проблемах и недостающих возможностях, которые могли бы ускорить темпы разработки. Для начала будет неплохо подумать о том, какой уровень взаимодействия с Kubernetes мы должны ожидать от наших разработчиков. В итоге мы уясним, насколько общими должны быть наши абстракции.

Спектр абстрагирования

В прошлом нам встречались несколько надменные высказывания вроде "Если вашим разработчикам известно о том, что они используют Kubernetes, это провал!" Такой взгляд на взаимодействие с Kubernetes может быть вполне приемлемым, особенно если вы создаете продукты или услуги, потребителям которых все равно, какая технология оркестрации находится внутри. Например, вы можете работать над системой управления базами данных, которая поддерживает разные БД. Ваших разработчиков, скорее всего, не интересует, внутри чего выполняются сегменты или экземпляры базы данных: Kubernetes, Bosh или Mesos! Однако делать эту философию основным критерием успеха вашей команды будет опасно. По мере создания поверх Kubernetes платформы с сервисами для лучшего обслуживания наших клиентов нам неоднократно придется принимать решения о том, как должны выглядеть наши абстракции. Это проиллюстрировано на рис. 1.9.

Данный вопрос может не давать покоя тем, кто занимается созданием платформы. Абстракции приносят много пользы. Такие проекты, как Cloud Foundry, предостав-

ляют разработчикам полнофункциональное окружение, например, с помощью одной единственной команды `cf push` мы можем взять приложение, собрать его и развернуть для обслуживания реального трафика. Сосредоточившись на достижении этой цели и описанных возможностей, Cloud Foundry углубляет свою поддержку работы поверх Kubernetes, и мы ожидаем, что этот переход будет выглядеть, скорее, как аспект реализации, чем как изменение функциональности. Еще одна закономерность, которую мы наблюдаем, состоит в желании предлагать компаниям нечто большее, чем просто Kubernetes, но при этом не вынуждать разработчиков делать сознательный выбор между разными технологиями. Например, в некоторых компаниях параллельно с Kubernetes используется Mesos. Такие компании создают абстракцию, позволяющую прозрачно выбирать место размещения приложений, не перекладывая эту обязанность на разработчиков, что также позволяет им избежать зависимости от конкретной технологии. Противоположный подход — создание абстракции поверх двух систем, которые работают по-разному. Это требует от компании зрелости в принятии решений и существенных инженерных усилий. К тому же, несмотря на то, что разработчикам больше не нужно знать, как взаимодействовать с Kubernetes или Mesos, они должны уметь работать с абстрагированной системой, существующей только в их компании. В наш век открытого исходного кода разработчики, какой бы стек технологий они ни использовали, не очень-то любят изучать системы, опыт работы с которыми будет бесполезен в других организациях.

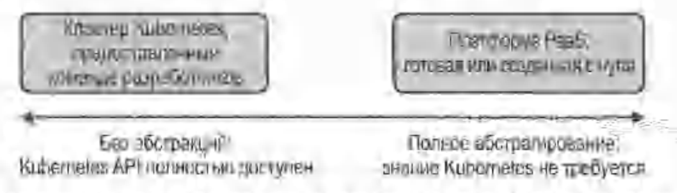


Рис. 1.9. Противоположные концы спектра: предоставление каждой команде отдельного кластера Kubernetes или полная инкапсуляция Kubernetes в виде платформы как сервиса (англ. Platform as a Service или PaaS)

Наконец, еще одна ловушка, которая нам встречалась, — "одержимость абстракциями", которая не дает предоставить доступ к ключевым возможностям Kubernetes. Со временем это может превратиться в игру в кошки-мышки, когда мы пытаемся не отставать от проекта, в результате чего наша абстракция может стать такой же сложной, как и система, которую мы абстрагируем.

На противоположном конце спектра находятся создатели платформ, которые хотят предложить разработчикам кластеры с самообслуживанием, которые тоже могут быть отличной моделью. Ответственность за взаимодействие с Kubernetes перекладывается на команды разработчиков. Понимают ли они, как работают Развертывания (Deployments), Наборы реплик (Replicas), Pod, Сервисы и API-интерфейсы объектов `Ingress`? Умеют ли они обращаться с единицами измерения вычислительных ресурсов, milliCPU, и знают ли они, как происходит выделение недоступных ресурсов (overcommit)? Известно ли им, как добиться того, чтобы выполнение приложений, сконфигурированных с более чем одной репликой, всегда планировалось на разных узлах? Если да, то это идеальная возможность избежать чрезмерного ус-

ложнения платформы приложений и позволить разработчикам создавать свои приложения прямо поверх Kubernetes.

Эта модель, в которой команды разработки сами отвечают за свои кластеры, встречается немного реже. Даже если команда состоит из людей, у которых есть опыт работы с Kubernetes, они, скорее всего, не захотят тратить время, в которое они могли бы заниматься реализацией возможностей, на определение того, как управлять жизненным циклом своего кластера Kubernetes при его обновлении. Kubernetes имеет чрезвычайно гибкую конфигурацию, но во многих случаях ожидать, что разработчики, занимающиеся созданием программного обеспечения, смогут еще овладеть Kubernetes на уровне специалистов, было бы нереалистично. Как вы сами увидите в следующих главах, решение об абстрагировании не обязательно должно быть категоричным. Определить, где необходимы абстракции, можно на разных этапах разработки. Мы будем стараться обеспечить подходящую степень гибкости и при этом помочь разработчикам добиваться своих целей.

Определение возможностей платформы

При создании платформы поверх Kubernetes необходимо определиться с тем, какие функции должны быть в нее встроены, а какие необходимо реализовать на уровне приложений. Обычно это решение принимается в каждом отдельном случае. Представьте, к примеру, что каждый микросервис Java реализует библиотеку для взаимодействия с другими сервисами по mTLS (mutual TLS). Таким образом, приложения получают средство идентификации приложений и шифрования данных по сети. Мы должны иметь глубокое понимание этого сценария использования, чтобы определить, следует ли его предоставлять или реализовывать на уровне платформы. Многие разработчики пытаются решить эту проблему путем встраивания в кластер технологии межсервисного взаимодействия (англ. service mesh). При анализе такого решения будут выявлены следующие факторы.

Преимущества внедрения mesh-сети:

- в приложения, написанные на Java, больше не нужно встраивать библиотеки для поддержки mTLS;
- приложения на других языках могут пользоваться той же системой mTLS/шифрования;
- упрощается задача, которую нужно решать разработчикам приложений.

Недостатки внедрения mesh-сети:

- организация mesh-сети — нетривиальная задача; это еще одна распределенная система, повышающая сложность эксплуатации;
- обычно возможности средств межсервисного взаимодействия далеко не ограничиваются идентификацией и шифрованием;
- у API-интерфейса идентификации в mesh-сети может отсутствовать интеграция с внутренней системой, которую используют уже имеющиеся приложения.

Взвесив указанные преимущества и недостатки, мы можем прийти к выводу о том, стоит ли решать эту проблему на уровне платформы. Главное — помнить, что нам не нужно перекладывать на платформу решение всех прикладных вопросов, и мы не должны к этому стремиться. Это один из многочисленных примеров поиска баланса, которые будут встречаться в нашей книге. Мы поделимся с вами несколькими советами, рекомендованными методиками и наставлениями, но, как и все остальное, вы должны оценивать их с учетом специфики вашей деятельности.

Составные компоненты

Давайте в заключение этой главы определим конкретные ключевые компоненты, которые вам будут доступны при построении платформы. Это касается как основополагающих частей системы, так и дополнительных сервисов, которые вы можете реализовать по своему желанию.

Компоненты, представленные на рис. 1.10, имеют различное значение для разных людей.

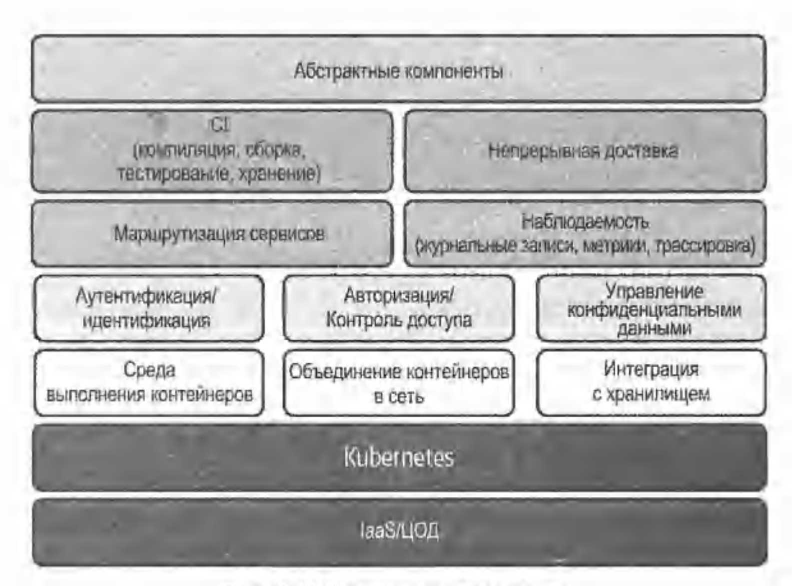


Рис. 1.10. Составные компоненты, используемые при создании платформы

Некоторые компоненты, такие, как средства организации сети контейнеров и среда их выполнения, должны быть в любом кластере (кластер Kubernetes, который не может выполнять приложения или не позволяет им взаимодействовать между собой, был бы не очень полезным). Вы, скорее всего, столкнетесь с тем, что некоторые компоненты иногда должны быть реализованы, а иногда нет. Например, управление секретами может не быть частью тех возможностей платформы, которые вы намереваетесь реализовать, если приложения уже используют для этого внешнее решение.

На рис. 1.10 явно недостает некоторых областей, таких, как безопасность. Дело в том, что безопасность — это не возможность, а, скорее, результат того, как все реализовано, начиная с уровня IaaS и выше. Давайте проведем общий обзор этих ключевых областей, а более подробное их рассмотрение отложим на потом.

IaaS/ЦОД и Kubernetes

IaaS/ЦОД и Kubernetes составляют основополагающий слой, о котором мы неоднократно упоминали в этой главе. Мы не хотим, чтобы вы принимали его как должное, так как от его стабильности будет напрямую зависеть устойчивость функционирования нашей платформы. Однако в современных реалиях мы проводим намного меньше времени за определением конфигурации наших серверных стоек для поддержки Kubernetes по сравнению с выбором из разнообразных вариантов и схем развертывания. В сущности, нам нужно уяснить, как мы будем подготавливать к работе и делать доступными кластеры Kubernetes.

Среда выполнения контейнеров

Среда выполнения контейнеров отвечает за управление жизненным циклом наших рабочих заданий на каждом хосте. Как правило, ее реализуют с помощью технологии, способной управлять контейнерами, такой как CRI-O, containerd или Docker. Выбор между данными вариантами возможен благодаря CRI (Container Runtime Interface — интерфейс среды выполнения контейнеров). Помимо этих популярных примеров существуют специализированные среды с поддержкой уникальных возможностей, таких как выполнение приложений на микро-ВМ.

Объединение контейнеров в сеть

Средства объединения контейнеров в сеть обычно отвечают за управление IP-адресами (англ. IP Address Management или IPAM) приложений и протоколы маршрутизации для обеспечения взаимодействия в сети. Распространенные технологии в этой области включают Calico и Cilium. Этот выбор стал возможным благодаря CNI (Container Networking Interface — интерфейс управления сетью контейнеров). Если подключить к кластеру технологию организации сети из контейнеров, агент kubelet сможет запрашивать IP-адреса рабочих заданий, которые он запускает. Некоторые подключаемые модули даже реализуют сервисы поверх сети подов.

Интеграция с хранилищем

Интеграция с хранилищем нужна в тех случаях, когда нам попросту недостаточно дискового накопителя на самом узле. Все больше и больше организаций, использующих современные версии Kubernetes, все чаще развертывают рабочие задания с сохранением состояния в своих кластерах. Эти задания требуют некоторой степени уверенности в том, что их состояние не будет утеряно в результате сбоя приложения или событий перепланирования. Хранилище может быть предоставлено распространенными системами, такими как vSAN, EBS, Ceph и многими другими.

Возможность выбора между разными реализациями обеспечивается за счет CSI (Container Storage Interface — интерфейс хранилищ для контейнеров); подобно CNI и CRI, он позволяет нам развернуть в нашем кластере подключаемый модуль, способный удовлетворить требования приложения к хранению данных.

Маршрутизация сервисов

Маршрутизация сервисов — это перенаправление трафика к приложениям, которые мы выполняем в Kubernetes, и от них. Kubernetes предлагает для этого Service API, но лишь в качестве отправной точки для поддержки более функциональных механизмов маршрутизации. Интерфейс Service API основан на сети контейнеров и предоставляет такие высокоуровневые возможности, как маршрутизация на прикладном уровне, шаблоны трафика и многое другое. Зачастую это реализуется с помощью технологии под названием контроллер Ingress. Пример использования более низкоуровневых возможностей маршрутизации сервисов — mesh-сеть. Эта технология имеет полноценную реализацию таких механизмов, как mTLS между сервисами и наблюдаемость, а также поддержку шаблонов проектирования уровня приложений, таких как "предохранитель".

Управление секретами

Здесь речь идет об администрировании и распространении секретов, которые нужны приложениям. Для этого у Kubernetes предусмотрен интерфейс Secrets API, через который можно взаимодействовать с подобного рода информацией. Однако многие кластеры не предлагают готовых решений для шифрования и управления конфиденциальными данными, наличия которых требуют различные организации. Это в основном касается вопросов углубленной защиты. На элементарном уровне мы можем гарантировать, что данные будут шифроваться перед сохранением (так называемое шифрование неактивных данных). Если требуется что-то посложнее, мы можем предоставить интеграцию с разными технологиями вроде Vault или Cyberark, предназначенными для управления секретами.

Идентификация

Здесь имеется в виду аутентификация пользователей и рабочих заданий. Первое, что спрашивают у многих администраторов кластеров, — как аутентифицировать пользователей с помощью LDAP или системы IAM облачного провайдера. Идентификация нужна не только людям, но и приложениям, например, для поддержки сетевых моделей с нулевым доверием, в которых намного сложнее выдать одно задание за другое. Этого можно достичь за счет интеграции провайдера идентификации и проверки подлинности приложений с помощью таких механизмов, как mTLS.

Авторизация/контроль допуска

Следующим шагом после идентификации пользователя или приложения является авторизация. Как мы должны выдавать или запрещать доступ к ресурсам, когда пользователи или приложения взаимодействуют с API-сервером? Kubernetes пред-

лагает поддержку RBAC в виде поля verbs внутри ресурсов, но как обеспечить нестандартную логику, необходимую для авторизации внутри нашей компании? Контроль допуска позволяет сделать еще один шаг в этом направлении и написать нашу собственную проверочную логику, которая в простейшем случае может заключаться в проверке статического списка правил, а в более сложном может выглядеть как динамическое обращение к другим системам для определения корректного ответа на запрос авторизации.

Цепочка поставки ПО

Цепочка поставки ПО охватывает весь путь программного обеспечения от исходного кода до среды выполнения. Это включает в себя распространенные вопросы, связанные с непрерывной интеграцией (англ. Continuous Integration или CI) и непрерывной доставкой (англ. Continuous Delivery или CD). Зачастую основная точка взаимодействия разработчиков — конвейеры (процессы), которые они конфигурируют в этих системах. Качественная интеграция систем CI/CD с Kubernetes может стать залогом успеха вашей платформы. Помимо этого также следует принимать во внимание такие факторы, как хранение артефактов, их безопасность с точки зрения уязвимостей и обеспечение целостности образов, которые будут выполняться в вашем кластере.

Наблюдаемость

Наблюдаемость — это обобщающий термин, относящийся ко всему, что помогает нам следить за происходящим в наших кластерах, как на системном, так и на прикладном уровне. Обычно считается, что наблюдаемость охватывает три ключевые области: журнальные записи, метрики и трассировку. Журналирование, как правило, заключается в передаче журнальных данных от рабочих заданий на хосте в целевую внутреннюю систему. В этой системе мы можем агрегировать и анализировать журнальные записи, переводя их в подходящий для потребления вид. Метрики — это данные, представляющие некое состояние в определенный момент времени. Мы обычно агрегируем или извлекаем эти данные в какую-то систему для дальнейшего анализа. Популярность трассировки во многом обусловлена потребностью в понимании взаимодействий между различными сервисами, из которых состоит стек нашего приложения. По мере сбора данных трассировки их можно направлять в систему агрегации, которая определяет время жизни запроса или ответа с помощью какого-то контекста или идентификатора корреляции.

Абстрактные компоненты

Абстрактные компоненты — это инструменты и сервисы, предназначенные для того, чтобы разработчики могли успешно пользоваться нашей платформой. Как уже обсуждалось ранее, существует целый спектр подходов к абстрагированию. Одни организации делают так, чтобы работа с Kubernetes была совершенно прозрачной для разработчиков. Другие решают сделать доступными многочисленные возможности Kubernetes, предоставляя каждому разработчику высокую степень гибкости.

Готовые решения также в основном ориентированы на процесс адаптации разработчиков, обеспечивая им доступ к среде и безопасные средства управления ею, которые они смогут применять в рамках платформы.

Резюме

В данной главе мы рассмотрели идеи, охватывающие Kubernetes, платформы приложений и даже построение таких платформ на основе Kubernetes. Надеемся, это дало вам пищу для размышлений о различных направлениях, изучение которых поможет вам лучше понять, как поверх этого замечательного средства оркестрации можно построить что-то новое. В оставшейся части книги мы подробно рассмотрим эти ключевые направления и поделимся нашими знаниями, случаями из жизни и советами, которые позволят вам сформировать собственные взгляды на процесс создания платформ. Давайте же, наконец, вступим на этот путь к эксплуатации!

ГЛАВА 2

Модели развертывания

Если мы хотим использовать Kubernetes в реальной среде, первый шаг очевиден: развернуть Kubernetes в нашей организации. Это включает в себя установку систем для создания кластеров Kubernetes и администрирования будущих обновлений. Поскольку Kubernetes — распределенная программная система, ее развертывание в основном сводится к процессу установки ПО. Однако важное отличие от многих других разновидностей программного обеспечения состоит в том, что платформа Kubernetes неразрывно связана с инфраструктурой. В связи с этим ее установка и подготовка инфраструктуры, на которую она будет устанавливаться, должны происходить одновременно.

В данной главе мы сначала рассмотрим предварительные вопросы развертывания кластеров Kubernetes и расскажем о том, насколько сильно стоит полагаться на управляемые сервисы и готовые продукты или проекты. Для тех, кто активно применяют существующие сервисы, продукты и проекты, бóльшая часть этой главы может быть не очень интересной, так как здесь мы обращаем основное внимание на процесс самостоятельной автоматизации. Представленный далее материал может пригодиться тем, кто анализирует средства развертывания Kubernetes, чтобы сориентироваться в разных доступных подходах. Для тех, кто находится в не совсем обычной ситуации и вынужден самостоятельно автоматизировать процесс развертывания Kubernetes, мы обсудим архитектурные аспекты общего характера, включая отдельные соображения по поводу etcd, а также поговорим об управлении различными кластерами. Мы также разберем полезные методы управления установленными копиями разнообразного ПО и инфраструктурными зависимостями, а также дадим обзор компонентов кластера и прольем свет на то, как они вместе работают. Еще в этой главе речь пойдет о способах управления дополнениями, которые устанавливаются поверх Kubernetes, а также о стратегиях обновления Kubernetes и дополнительных компонентов, из которых состоит ваша платформа приложений.

Управляемые сервисы и самостоятельное развертывание

Прежде чем углубляться в тему моделей развертывания, следует поговорить о том, нужна ли вам *в принципе* полноценная модель такого рода для Kubernetes. Облачные провайдеры предлагают управляемые сервисы, которые во многом облегчают процесс развертывания. Вам по-прежнему нужно разработать надежную декларативную систему для создания этих управляемых кластеров Kubernetes, однако инкапсуляция большинства подробностей о том, *как* развертывается кластер, может быть полезной.

Управляемые сервисы

Причина использования управляемых сервисов Kubernetes сводится к облегчению труда инженеров. Корректное управление развертыванием и жизненным циклом Kubernetes требует существенных усилий, связанных с проектированием и реализацией. И не забывайте, что Kubernetes является лишь одним из компонентов вашей платформы приложений, — оркестратором контейнеров.

Управляемый сервис, в сущности, предоставляет плоскость управления (англ. control plane) Kubernetes, к которой при желании можно присоединять рабочие узлы. Обязательства по масштабированию и обеспечению доступности снижаются плоскостью управления. Оба эти фактора важны. Более того, если вы уже используете готовые сервисы облачного провайдера, вам будет еще проще. Например, если вы работаете в AWS (Amazon Web Services) и уже применяете Fargate для бессерверных вычислений, IAM (Identity and Access Management — управление идентификацией и доступом) для контроля доступа на основе ролей и CloudWatch для наблюдаемости, то вы можете пользоваться этими сервисами в сочетании с EKS (Elastic Kubernetes Service) для решения различных задач в своей платформе приложений.

Это чем-то похоже на управляемые сервисы БД. Если вас в первую очередь интересует создание приложения, удовлетворяющего нуждам вашей организации, и для этого вам требуется реляционная база данных, но вы не можете оправдать наличие в своей команде отдельного администратора, который будет ее обслуживать, то использование платной БД, предоставляемой облачным провайдером, может оказаться чрезвычайно полезным. Вы сможете быстрее начать работу над приложением, а провайдер управляемых сервисов сам позаботится о доступности, резервном копировании и обновлениях. Во многих случаях это дает чистый выигрыш. Но у всякой медали есть обратная сторона.

Самостоятельное развертывание

За экономию на использовании управляемого сервиса Kubernetes приходится платить нехваткой гибкости и свободы. Это частично связано с риском попасть в зависимость от поставщика. Управляемые сервисы, как правило, предоставляются провайдерами облачной инфраструктуры. Если вы начнете активно пользоваться их услугами, проектируемые вами системы и сервисы, которые вы применяете, с большой долей вероятности не будут нейтральными к поставщикам. Если в будущем ваш поставщик поднимет цены или допустит снижение качества услуг, вы окажетесь загнанными в угол. Вы платили специалистам за решение вопросов, на которые у вас не было времени, и теперь эти специалисты обладают слишком большим влиянием на вашу дальнейшую судьбу.

Конечно, вы можете разложить яйца по разным корзинам, пользуясь услугами нескольких провайдеров, разница в том, какие возможности Kubernetes они предоставляют и как они это делают, может привести к неловким несоответствиям, с которыми нужно будет бороться.

В связи с этим у вас может возникнуть желание развернуть Kubernetes самостоятельно. Kubernetes предоставляет широкие возможности по конфигурации. Такая способность настройки делает платформу удивительно гибкой и мощной. Если вы потратите время на изучение и самостоятельное администрирование Kubernetes, весь мир платформ приложений будет у ваших ног. Нет такой функции, которую вы бы не смогли реализовать, и такого требования, которое вы бы не смогли удовлетворить. И ваши реализации будут совместимы с любыми провайдерами инфраструктуры, будь то общедоступные облачные сервисы или ваши собственные серверы, размещенные в частном центре обработки данных. Если вы учтете все инфраструктурные несоответствия, ваша платформа сможет всегда предоставлять доступ к одному и тому же набору возможностей Kubernetes. И разработчикам, которые пользуются вашей платформой, будет все равно, кто предоставляет внутреннюю инфраструктуру (они могут об этом даже не знать).

Просто помните, что разработчиков интересуют только возможности платформы, а не ее внутренняя инфраструктура или то, кто ее предоставляет. Если вы сами определяете набор доступных возможностей, и этот набор одинаков для всех провайдеров инфраструктуры, то у вас есть возможность обеспечить своим разработчикам превосходный опыт взаимодействия. Вы сможете выбрать подходящую версию Kubernetes. У вас будет доступ ко всем флагам и функциям компонентов плоскости управления, а также к оборудованию, программному обеспечению, которое на нем установлено и статическим манифестам Pod'ов, записанным на диск. В вашем распоряжении будет мощный и опасный инструмент, с помощью которого вы можете попытаться завоевать благосклонность своих разработчиков. Но не забывайте, что данный инструмент необходимо как следует изучить, иначе вы рискуете навредить как себе, так и другим.

Принятие решения

Путь к славе редко кажется ясным в самом начале. Если вы выбираете между управляемым сервисом Kubernetes и развертыванием собственных кластеров, значит, триумфальный финал еще далек. И выбранное решение будет иметь далеко идущие последствия для вашей организации. Вот несколько руководящих принципов, которые должны облегчить этот процесс.

Вы должны склоняться к управляемому сервису, если:

- идея изучения Kubernetes кажется слишком сложной;
- вы не хотите брать на себя ответственность за управление распределенной программной системой, которая является залогом успеха вашей компании;
- неудобства, связанные с ограниченными возможностями, которые предоставляются поставщиком, выглядят приемлемыми;
- вы верите в то, что поставщик вашего управляемого сервиса будет удовлетворять ваши потребности и оставаться хорошим бизнес-партнером.

Вы должны склоняться к самостоятельному развертыванию Kubernetes, если:

- вас беспокоят ограничения, накладываемые поставщиком;
- у вас практически нет веры в корпоративных гигантов, которые предоставляют инфраструктуру для облачных вычислений;
- вы в восторге от того, насколько мощную платформу можно создать на основе Kubernetes;
- вам не терпится задействовать это потрясающее средство оркестрации контейнеров, чтобы предоставить своим разработчикам чрезвычайно удобную платформу.

Если вы решите использовать управляемый сервис, то можете пропустить большую часть оставшегося в этой главе материала. К выбранному вами сценарию будут применимы только разделы "Дополнения" и "Механизмы автоматического срабатывания". Если же вы собираетесь управлять собственными кластерами, читайте дальше! В следующих разделах мы подробней обсудим модели развертывания и инструменты, на которые следует обратить внимание.

Автоматизация

Если вы решили заняться разработкой модели развертывания для своих кластеров Kubernetes, то первостепенное значение получает тема автоматизации. Это должно быть руководящим принципом любой такой модели. Устранение человеческого фактора — ключевое условие снижения расходов и повышения стабильности. Люди обходятся дорого. Вместо того чтобы оплачивать труд инженера, который выполняет рутинные, скучные действия, лучше инвестировать в инновации. Более того, на людей нельзя положиться. Они ошибаются. Всего лишь одна ошибка в последовательности шагов может сделать систему нестабильной или даже вывести ее из строя. Инженерные ресурсы, потраченные на автоматизацию развертываний с помощью программных систем, окупят себя сторицей, экономя человеческий труд и облегчая диагностику неисправностей.

Если вы решите взять на себя управление жизненным циклом кластера, то необходимо сформулировать стратегический план дальнейших действий. У вас есть выбор: воспользоваться готовым установщиком Kubernetes или разработать свои собственные средства автоматизации с нуля. Это решение перекликается с выбором между использованием управляемого сервиса и развертыванием своего собственного. Один вариант даст вам большие возможности, контроль и гибкость, но за счет увеличения трудозатрат со стороны инженеров.

Готовый установщик

В настоящее время существует очень много установщиков Kubernetes, как с открытым исходным кодом, так и с поддержкой со стороны компаний-разработчиков. Для наглядности: на веб-сайте CNCF (**https://www.cncf.io/certification/kcsp**) сейчас насчитывается 180 сертифицированных поставщиков услуг Kubernetes. Некоторые

из них платные и сопровождаются услугами опытных инженеров, которые помогут вам подготовить ваши кластеры к работе, и к которым при необходимости можно будет обращаться в дальнейшем за поддержкой.

Для изучения и применения других установщиков придется провести исследования и поэкспериментировать. Некоторые из них (обычно те, за которые вы платите) позволяют установить Kubernetes с нуля одним нажатием кнопки. Если вам подходят указанные предписания и доступные параметры, и если вы можете позволить себе такие расходы, то данный метод установки может быть отличным выбором. На момент написания этих строк мы чаще всего встречаем в реальных условиях именно готовые установщики.

Собственные средства автоматизации

Собственные средства автоматизации обычно требуются в той или иной степени даже при наличии готового установщика. Это, как правило, принимает форму интеграции с существующими системами разработчиков. Однако в данном разделе речь пойдет о создании собственного установщика Kubernetes.

Если вы только начинаете знакомство с Kubernetes или меняете направление своей стратегии в отношении этой платформы, то самостоятельная автоматизация, скорее всего, будет оправданна только при выполнении всех перечисленных далее условий:

- ресурсы, которые вы можете выделить для этой цели, не ограничиваются одним или двумя инженерами;
- в вашей организации есть инженеры с огромным опытом работы с Kubernetes;
- у вас есть особые требования, которые не может как следует удовлетворить ни один управляемый сервис или готовый установщик.

Бо́льшая часть материала, оставшегося в этой главе, предназначена для одной из двух категорий читателей:

- тех, кто удовлетворяет всем требованиям для создания собственных средств автоматизации;
- тех, кто анализирует чужие установщики и хочет лучше понять, как выглядят удачные решения.

Это подводит нас к деталям создания собственных средств автоматизации для установки и администрирования кластеров Kubernetes. В основе всех этих вопросов должно быть четкое понимание задач вашей платформы, которые в свою очередь опираются на требования ваших разработчиков приложений (особенно тех, которые начнут пользоваться вашей платформой в числе первых). Создание платформы "в вакууме", без тесного взаимодействия с теми, для кого она предназначена, будет ошибкой. Сделайте первые, предварительные, версии доступными группам разработчиков, чтобы те смогли их протестировать. Организуйте продуктивный цикл обратной связи для устранения ошибок и добавления возможностей. От этого зависит успешное внедрение вашей платформы.

Дальше мы рассмотрим аспекты архитектуры, которые следует принять во внимание еще до начала реализации. Это касается моделей развертывания для etcd, разделения окружений развертывания на разные уровни, решения проблем с управлением большим количеством кластеров и выбора подходящих пулов узлов для размещения ваших рабочих заданий. Затем мы обсудим детали установки Kubernetes с учетом инфраструктурных зависимостей, программного обеспечения, инсталлированного на виртуальном или физическом оборудовании кластера, и, наконец, контейнерных компонентов, из которых состоит плоскость управления кластера Kubernetes.

Архитектура и топология

Этот раздел посвящен архитектурным решениям, имеющим далеко идущие последствия для систем, с помощью которых выделяются и администрируются кластеры Kubernetes. Это включает в себя модель развертывания etcd и уникальные факторы, которые необходимо принимать во внимание при работе с этим компонентом платформы. В число этих тем входят методы разделения различных кластеров на уровни в зависимости от целей уровня обслуживания (англ. Service Level Objective или SLO), которые перед ними стоят. Мы также рассмотрим концепцию группирования узлов по пулам с учетом того, для каких целей они используются в заданном кластере. В завершение мы обсудим способы федеративного управления кластерами и программным обеспечением, которое в них развернуто.

Модели развертывания etcd

Концепция etcd играет роль базы данных для объектов кластера Kubernetes и заслуживает особого внимания. Это распределенное хранилище данных, которое использует алгоритм консенсуса для хранения копии состояния вашего кластера на нескольких компьютерах. Узлы в кластере etcd должны быть организованы так, чтобы надежно поддерживать с помощью сетевых соединений. При разработке топологии сети нужно учитывать требования к сетевым задержкам (latency). Мы рассмотрим эту тему в данном разделе, а также затронем два основных архитектурных решения, которые необходимо выбрать при разработке модели развертывания для etcd: альтернатива между отдельной и совмещенной инфраструктурой, а также между выполнением внутри контейнера и установкой непосредственно на хосте.

Сетевые условия

Параметры, которые приняты в etcd по умолчанию, рассчитаны на задержки внутри одного центра обработки данных. Если распределить etcd между несколькими ЦОД, то следует проверить среднюю задержку приема-передачи между узлами и при необходимости отрегулировать интервал переключения и время ожидания выбора ведущих узлов. Мы настоятельно советуем вам отказаться от использования кластеров etcd, распределенных между регионами. Если для обеспечения повышенной доступности нужно несколько центров обработки данных, то они по крайней мере, должны находиться в непосредственной близости друг от друга в пределах одного региона.

Отдельная или совмещенная инфраструктура

Когда речь заходит о процессе развертывания, нас очень часто спрашивают, как размещать etcd: на отдельных серверах или совместно с другими компонентами плоскости управления, такими как API-сервер, планировщик, диспетчер контроллеров и т. д. Первое, о чем следует подумать, — это размер кластера, который вы будете администрировать, т. е. число рабочих узлов, составляющих кластер. Как подобрать оптимальный размер, мы обсудим позже в этой главе. Данный фактор во многом определяет, нужно ли выделять для etcd отдельные серверы.

Очевидно, что etcd — это очень важный компонент. Ухудшение его производительности отрицательно скажется на вашей способности управлять ресурсами кластера. Если ваши рабочие задания не зависят от Kubernetes API, с ними все должно быть в порядке, но поддержание плоскости управления в рабочем состоянии все равно имеет больше значение.

Если вы ведете машину, и у нее перестал работать руль, вас вряд ли утешит тот факт, что машина все еще едет по дороге. Напротив, такая ситуация может быть крайне опасной. По этой причине, если вы предъявляете к etcd требования по вводу/выводу, рассчитанные на крупные кластеры, будет разумно выделить отдельные серверы, чтобы не соперничать за ресурсы с другими компонентами плоскости управления. В данном контексте понятие "крупного кластера" зависит от того, сколько компьютеров использует плоскость управления, но в любом случае речь идет не меньше, чем о 50 рабочих узлах. Если вы занимаетесь планированием для кластеров с более чем 200 рабочими узлами, лучше сразу ориентироваться на выделенные кластеры etcd. Если же речь идет о небольших кластерах, лучше будет совмещенный вариант, так как накладные расходы и стоимость инфраструктуры снизятся. Вы, скорее всего, воспользуетесь Kubeadm, инструментом для начальной конфигурации Kubernetes, который поддерживает эту модель и позаботится о сопутствующих проблемах.

Размещение: в контейнере или на хосте

Следующий распространенный вопрос состоит в том, устанавливать etcd на компьютере или выполнять внутри контейнера. Давайте сначала рассмотрим простой ответ: если etcd работает в совмещенной реализации, применяйте контейнеры. Такой вариант будет лучшим для начальной конфигурации Kubernetes. Но, если вы решили разместить etcd на отдельных серверах, то у вас есть следующий выбор: etcd можно установить на хосте, что позволяет встроить этот компонент в образы систем и избежать дополнительных трудностей, связанных с установкой среды выполнения контейнеров. Но, если etcd работает в контейнере, то самое разумное решение — установить в системе среду выполнения контейнеров и kubelet и запускать etcd с помощью статического манифеста. Преимущество подобного способа состоит в унификации подходов и методов установки всех компонентов плоскости управления. Повторное применение одних и тех же методик в сложных системах приносит пользу, но во многом зависит от ваших предпочтений.

Уровни кластера

Разделение кластеров на уровни мы наблюдаем почти повсеместно. Эти уровни зачастую относятся к тестированию, разработке, финальному тестированию и развертыванию в условиях эксплуатации. Некоторые разработчики называют это разными "окружениями". Но данный термин носит общий характер и может иметь разные значения. Здесь мы будем употреблять термин *уровень*, чтобы выделить разные типы кластеров. В частности, речь пойдет об SLO (англ. Service Level Objective — цель уровня обслуживания) и SLA (англ. Service Level Agreement — соглашение об уровне обслуживания), которые могут быть связаны с кластером, а также о назначении кластера и его роли в подготовке приложения к реальной среде (если ему таковая отведена). То, как именно будут выглядеть эти уровни, зависит от особенностей организации, но есть некоторые общие закономерности, и мы объясним, что обычно означает каждый из этих четырех распространенных уровней. На всех уровнях должны использоваться одни и те же системы развертывания кластера и управления жизненным циклом. Если их активно применять на более низких уровнях, то они всегда будут предсказуемо работать и в важных промышленных кластерах.

- *Тестирование*. Кластеры на уровне тестирования выделены отдельно и носят временный характер. К ним нередко применяется параметр TTL (time-to-live — срок жизни), чтобы они автоматически уничтожались по окончании заданного отрезка времени (обычно не дольше недели). Разработчики платформы очень часто развертывают такие кластеры для тестирования отдельных ее компонентов или функций, над которыми идет работа. Уровни тестирования также целесообразны в случаях, когда разработчикам не хватает локального кластера, или в качестве еще одного этапа тестирования, который идет вслед за выполнением тестов на локальном кластере. Это происходит особенно часто, когда разработчики приложения начинают упаковывать свой код в контейнеры и тестировать его в Kubernetes. Для этих кластеров не предусмотрены SLO или SLA. Они используют последнюю версию платформы или, при необходимости, ту, что находится на начальной стадии разработки (pre-alpha).
- *Разработка*. Кластеры уровня разработки, как правило, "постоянные" и не имеют TTL. Они работают в мультитенантном режиме (mutli-tenancy) там, где это уместно, и обладают всеми возможностями промышленного кластера. Их используют для первого этапа интеграционного тестирования приложений и проверки на совместимость с альфа-версиями платформы. Кластеры этого уровня также применяются для общего тестирования и разработки приложений. У них обычно есть SLO, но отсутствует формальное соглашение об уровне услуг. По своим требованиям к доступности они зачастую лишь немногим уступают производственным кластерам, по крайней мере, в рабочее время, так как перебои в работе влияют на продуктивность разработчиков. Для сравнения: приложения, выполняющиеся в кластерах уровня разработки, вообще не имеют никаких SLO или SLA, они очень часто обновляются и постоянно претерпевают изменения. Именно для таких кластеров официально выпускается альфа- или бета-версия платформы.

- *Финальное тестирование.* Кластеры уровня финального тестирования, как и в предыдущем случае, являются постоянными и нередко задействуются в мультитенантном режиме. Они предназначены для выполнения заключительных интеграционных тестов и одобрения перед развертыванием в реальных условиях. Их используют заинтересованные стороны, не принимающие непосредственного участия в разработке ПО, которое там выполняется. Это касается руководителей проекта, владельцев продукта и руководства компании. В некоторых случаях речь может идти о клиентах или внешних заинтересованных сторонах, которым нужен доступ к еще не вышедшим версиям программного обеспечения. SLO у таких кластеров зачастую похожа на ту, которая применяется на уровне разработки. Уровни финального тестирования имеют формальное SLA, если у внешних заинтересованных сторон или платных клиентов есть доступ к рабочим заданиям кластера. Кластеры такого рода выполняют официально выпущенную бета-версию платформы, если в ней строго соблюдается обратная совместимость. В случае, если обратная совместимость не гарантирована, кластеры финального тестирования должны выполнять ту же стабильную версию платформы, которая развернута в промышленных условиях.
- *Производственное окружение.* Кластеры уровня эксплуатации являются источником прибыли. Они предназначены для взаимодействия с клиентами и выполнения коммерческих приложений и веб-сайтов. Здесь применяются только те выпуски ПО, которые одобрены и готовы к внедрению. Версии платформы тоже должны быть одобренными и стабильными. Эти кластеры имеют подробные, четко определенные SLO, а зачастую и SLA, обладающие юридической силой.

Пулы узлов

Пулы узлов позволяют группировать узлы отдельно взятого кластера по типам, которые могут отличаться какими-то уникальными характеристиками или предназначением. Прежде чем углубляться в подробности, следует разобраться в компромиссе, который связан с наличием пула узлов: выбор между использованием пулов с множеством узлов в рамках одного или нескольких разных кластеров. Для таких пулов у ваших приложений должны быть поля `nodeSelector`, чтобы они размещались в подходящем пуле. Вам также, скорее всего, понадобятся ограничения узлов (taints), чтобы приложения без поля `nodeSelector` не размещались случайно там, где не следует. Кроме того, данный подход усложняет масштабирование узлов внутри кластера, так как вашим системам придется отслеживать разные пулы и масштабировать их по отдельности. С другой стороны, если каждый пул находится в своем кластере, эти вопросы перекладываются на средства управления кластером и межкластерного взаимодействия ПО. Вам понадобится больше кластеров. И вам придется эффективно распределять приложения между кластерами. Эти преимущества и недостатки объединения узлов в пулы сведены в табл. 2.1.

Таблица 2.1. Преимущества и недостатки пулов узлов

Преимущества	Недостатки
Меньше кластеров, которые нужно администрировать	Для приложений часто требуются поля `nodeSelector`
Меньше кластеров для размещения приложений	Необходимо применять ограничения узлов и управлять ими
—	Усложняются процедуры масштабирования кластеров

Пул, основанный на характеристиках, состоит из узлов с компонентами или атрибутами, которые нужны или подходят для определенных приложений. Пример — наличие такого специализированного устройства, как графический адаптер (англ. Graphics Processing Unit или GPU). Еще одним примером характеристики может быть тип сетевого интерфейса, используемого узлом, или соотношение памяти и вычислительных ресурсов компьютера. Причины, по которым стоит отдать предпочтение узлам с разными соотношениями этих ресурсов, будут подробно рассмотрены в разделе "Инфраструктура" далее в этой главе. А пока достаточно сказать, что все эти характеристики нацелены на разные типы приложений, и, если их совместить в одном кластере, потребуется группировка в пулы, чтобы иметь возможность управлять размещением разных Pod'ов.

Пул узлов, основанный на ролях, выполняет конкретную функцию, и его часто следует оградить от конкуренции за ресурсы. Узлы, попавшие в такой пул, могут не иметь каких-то определенных характеристик, важно только их назначение. Распространенный пример — выделение пула узлов для входного слоя кластера. В этом случае отдельный пул не только защищает рабочие задания от конкуренции за ресурсы (в данном примере это особенно важно, так как в настоящее время к использованию сети нельзя применять запросы ресурсов и лимиты), но и упрощает сетевую модель и отдельные узлы, принимающие трафик от источников за пределами кластера. В отличие от пулов, основанных на характеристиках, эти роли зачастую нельзя разделить на отдельные кластеры, так как оборудование имеет большое значение для работы конкретного кластера. Тем не менее, если вы группируете узлы по пулам, то для этого должна быть веская причина. Не создавайте пулы без разбору. Кластеры Kubernetes и так достаточно сложны. Не усложняйте себе жизнь без необходимости.

Имейте в виду, что вам, скорее всего, придется решать проблемы с администрированием множества разных кластеров независимо от того, есть ли пулы узлов или нет. Очень сложно найти организацию, которая использует Kubernetes, но не имеет большого количества отдельных кластеров. Для этого есть множество разнообразных причин. Поэтому, если вам захотелось применить пулы узлов на основе характеристик, подумайте над тем, чтобы направить инженерные усилия на разработку и усовершенствование средств управления многокластерными системами. В результате вы получите возможность легко выделять кластеры для оборудования с разными характеристиками, которое вам необходимо предоставлять.

Федерация кластеров

Федерация — это общее понятие, обозначающее метод централизованного управления всеми кластерами, которые находятся под вашим контролем. Kubernetes чем-то напоминает запретный плод. Когда вы осознáете, насколько он нравится, сразу захочется больше. Но, по той же аналогии, если не контролировать свои желания, можно наломать дров. Стратегии федерации позволяют предприятиям управлять своими программными зависимостями так, чтобы они не стали затратными и пагубными.

Распространенный и полезный подход — объединение кластеров в федерацию сначала по регионам, а затем глобально. Это уменьшает масштабы потенциальных проблем и снижает вычислительную нагрузку на данные кластеры. Приступая к созданию федерации кластеров, вы можете не обладать глобальным присутствием или объемом инфраструктуры, которые бы оправдывали многоуровневый федеративный подход, но в будущем такой шаг может потребоваться, поэтому помните о нем как об архитектурном принципе.

Давайте обсудим некоторые важные вопросы из этой области. В данном разделе речь пойдет о том, как управляющие кластеры могут помочь с консолидацией и централизацией региональных сервисов. Мы рассмотрим, как можно консолидировать метрики рабочих заданий в различных кластерах, и как это влияет на управление приложениями, развернутыми в разных кластерах с применением централизованного управления.

Управляющие кластеры

Назначение управляющих кластеров понятно из их названия: они управляют другими кластерами Kubernetes. По мере того, как число кластеров, администрируемых организацией, растет, и их использование становится все активней, возникает необходимость в программных системах для облегчения их эксплуатации. И, как можно было бы ожидать, для выполнения этих систем нередко выбирают платформы на основе Kubernetes. Популярным проектом для этого стал Cluster API (**https://cluster-api.sigs.k8s.io**). Это набор операторов Kubernetes, которые представляют другие кластеры Kubernetes и их компоненты с помощью нестандартных ресурсов вроде Cluster и Machine. Компоненты Cluster API обычно размещают в управляющем кластере для развертывания и администрирования инфраструктуры других кластеров.

Однако функционирование управляющего кластера в таком качестве имеет свои недостатки. Обычно вопросы, относящиеся к уровню эксплуатации, целесообразно отделять от других уровней. Следовательно, у многих организаций будет управляющий кластер, предназначенный специально для реальных условий. Это увеличивает накладные расходы, связанные с управляющим кластером. Еще одна проблема состоит в автомасштабировании кластеров — автоматическом добавлении и удалении экземпляров приложений в ответ на изменение нагрузки. Cluster Autoscaler (механизм автомасштабирования кластеров) обычно работает внутри кластера, масштабированием которого он занимается, чтобы следить за условиями,

требующими изменения числа узлов. Однако за выделение и удаление этих узлов отвечает контроллер, содержащийся в управляющем кластере. Это превращает последний в дополнительную зависимость для любого кластера с приложениями, который использует Cluster Autoscaler, как проиллюстрировано на рис. 2.1. Как быть, если управляющий кластер станет недоступным в момент, когда вашему кластеру нужно масштабироваться для удовлетворения спроса?



Рис. 2.1. Cluster Autoscaler обращается к управляющему кластеру, чтобы инициировать масштабирование

Чтобы исправить ситуацию, компоненты Cluster API можно разместить вместе с приложениями и сделать их автономными. В этом случае ресурсы Cluster и Machine будут размещаться в том же кластере. Управляющий кластер по-прежнему можно задействовать для создания и удаления других кластеров, но кластеры с приложениями становятся во многом самодостаточными и больше не полагаются на него при выполнении таких рутинных операций, как автомасштабирование (рис. 2.2).

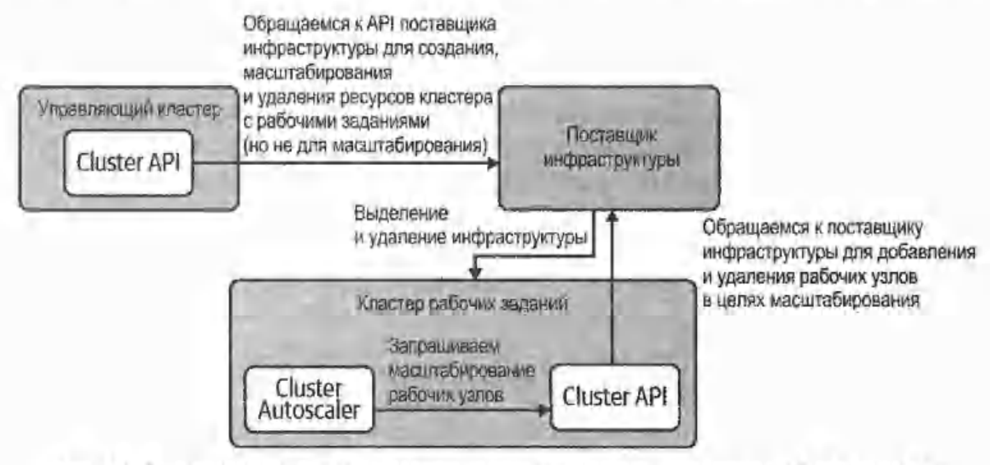


Рис. 2.2. Cluster Autoscaler обращается к компоненту локального интерфейса Cluster API для выполнения масштабирования

Явное преимущество описанного подхода состоит в том, что, если любым другим контроллерам или приложениям в кластере понадобятся метаданные или атрибуты, содержащиеся в нестандартных ресурсах Cluster API, они могут их получить, про-

читав эти ресурсы посредством локального API-интерфейса. Для этого им не нужно обращаться к API-интерфейсу управляющего кластера. Например, если у вас есть контроллер Namespace, поведение которого меняется в зависимости от того, в каком кластере он выполняется (в режиме эксплуатации или том, который предназначен для разработки), эта информация уже может содержаться в ресурсе Cluster, представляющем его собственный кластер.

В управляющих кластерах нередко дополнительно размещаются разделяемые или региональные сервисы, к которым обращаются системы из ряда других кластеров. Это нельзя отнести к функциям *управления*. Многие управляющие кластеры просто являются подходящим местом для выполнения этих разделяемых сервисов, примерами которых могут служить системы CI/CD и реестры контейнеров.

Наблюдаемость

Одна из непростых задач администрирования большого числа кластеров — сбор метрик по всей этой инфраструктуре и хранение их (или их подмножества) в центральном месте. Общие параметры, поддающиеся измерению и дающие четкое представление о работоспособности кластеров и рабочих заданий, которыми вы управляете, являются важным аспектом федерации кластеров. Prometheus (**https://prometheus.io**) — это зрелый проект с открытым исходным кодом, с помощью которого многие организации собирают и хранят метрики. Даже если вы его не используете, модель, применяемая им для федерации, стоит того, чтобы ее изучить и по возможности воспроизвести с применением имеющихся инструментов. Проект поддерживает региональный подход, позволяя объединенным в федерацию серверам Prometheus извлекать подмножества метрик из других, более низкоуровневых серверов Prometheus. Таким образом он будет совместим с любой стратегией создания федерации, которой вы привержены. Эта тема подробно исследуется в *главе 9*.

Федеративное развертывание программного обеспечения

Еще один важный вопрос при администрировании разных удаленных кластеров заключается в том, как управлять развертыванием ПО в этих кластерах. Одно дело управлять кластерами самостоятельно, и совсем другое — организовать развертывание в этих кластерах приложений конечных пользователей. Для этого, в конце концов, и нужны все кластеры. У вас, наверное, есть важные и ценные приложения, которые необходимо развертывать в разных регионах, чтобы обеспечить их доступность. Или, возможно, вам просто нужно сделать так, чтобы приложения развертывались с учетом характеристик различных кластеров. Судя по отсутствию единогласия относительно того, как лучше решать эту проблему, принимать такие решения нелегко.

Сообщество Kubernetes пытается бороться с указанной проблемой с помощью метода, который уже на протяжении некоторого времени является широко применимым. Самым свежим воплощением этого подхода служит проект KubeFed (**https://github.com/kubernetes-sigs/kubefed**). Он также применяется для управления конфигурацией кластеров, но здесь нас больше интересуют определения приложений, предназначенных для нескольких кластеров. Один из полезных архитек-

турных принципов, который возник благодаря ему, состоит в возможности объединения в федерацию любых типов, применяемых в API-интерфейсе Kubernetes. Например, вы можете использовать федеративные версии типов Namespace и Deployment, чтобы описать, как должны применяться ресурсы к разным кластерам. Это мощный принцип в том смысле, что вы можете создать FederatedDeployment централизованным образом в одном управляющем кластере, в результате чего на основе этого манифеста в других кластерах будет создано несколько удаленных объектов Deployment. Однако мы ожидаем, что в будущем прогресс в этой области продолжится. На сегодня самым распространенным решением, которое нам встречается в реальных условиях, по-прежнему является конфигурирование средств CI/CD для развертывания рабочих заданий на разных кластерах.

Итак, мы обсудили широкий спектр архитектурных вопросов, которые будут определять способ организации и администрирования ваших кластеров. Теперь давайте подробно рассмотрим вопросы, связанные с архитектурой.

Инфраструктура

Развертывание Kubernetes — это процесс установки ПО, который зависит от ИТ-инфраструктуры. Кластер Kubernetes можно развернуть на личном ноутбуке с помощью виртуальных машин или контейнеров Docker. Но это будет лишь имитация, предназначенная для тестирования. При внедрении в эксплуатацию необходимо наличие различных компонентов инфраструктуры, и они зачастую выделяются в рамках развертывания самой платформы Kubernetes.

Кластер Kubernetes, пригодный к использованию в режиме эксплуатации, должен состоять из некоторого числа компьютеров, подключенных к сети. Чтобы избежать путаницы с терминологией, мы будем называть эти компьютеры *серверами*. Эти серверы могут быть виртуальными (виртуальные машины) или физическими. Важно то, что у вас есть возможность их выделять, и главный вопрос состоит в том, каким образом они создаются или включаются.

Возможно, вам придется приобрести соответствующее оборудование и разместить его в центре обработки данных. Или же вы можете просто запросить ресурсы у облачного провайдера, и тот запустит нужные вам серверы. Каким бы ни был этот процесс, вам понадобятся серверы и правильно сконфигурированная сеть, и эти требования должны быть учтены в вашей модели развертывания.

Важный аспект действий, направленных на автоматизацию, — тщательное продумывание того, как автоматизировать управление инфраструктурой. Старайтесь избегать таких операций, как ручное заполнение форм в веб-интерфейсах. Отдавайте предпочтение декларативным инструментам, которые достигают того же результата путем обращения к API-интерфейсу. При подобной модели автоматизации нужно обеспечить возможность выделять серверы, сети и сопутствующие ресурсы по требованию, как это делают такие облачные провайдеры, как Amazon Web Services, Microsoft Azure или Google Cloud Platform. Однако не всякое окружение предоставляет API или пользовательский веб-интерфейс для выделения инфраструктуры.

Большое количество приложений работает в центрах обработки данных с множеством серверов, приобретенных и установленных теми же компаниями, которые их используют. Данный этап должен быть завершен задолго до установки и выполнения программных компонентов Kubernetes. Важно различать эти два процесса, чтобы определить те модели и методы, которые хорошо подходят для каждого из них.

В следующем разделе речь пойдет о трудностях развертывания Kubernetes на физическом оборудовании по сравнению с применением виртуальных машин для узлов самих кластеров. Затем мы обсудим выбор размера кластера и влияние на управление его жизненным циклом. Далее пройдемся по аспектам вычислительной и сетевой инфраструктуры, на которые следует обращать внимание. В итоге мы сформулируем определенные стратегии автоматизации управления инфраструктурой в кластерах Kubernetes.

Физическое и виртуальное оборудование

В ходе исследования Kubernetes многие задумываются о том, не утратил ли свою актуальность слой виртуальных машин. Разве контейнеры делают не то же самое? Получается, что вы используете два слоя виртуализации? Ответ: необязательно. Проекты, включающие Kubernetes, могут иметь огромный успех как в физических, так и в виртуальных окружениях. Выбор подходящей среды для развертывания крайне важен и должен делаться с оглядкой на то, какие проблемы решают те или иные технологии, и насколько уверенно этими технологиями владеет ваша команда.

Революция, произошедшая в сфере виртуализации, изменила концепцию выделения и администрирования нашей инфраструктуры. Раньше команды, обслуживавшие инфраструктуру, применяли такие методы, как загрузка хостов с помощью PXE, управление конфигурацией серверов и предоставление серверам резервного оборудования (хранилища). Современные виртуальные окружения прячут все это за API-интерфейсами, которые позволяют выделять, изменять и удалять ресурсы по желанию, даже не догадываясь о том, какое аппаратное обеспечение находится внутри. Эта модель доказала свою состоятельность в разных центрах обработки данных с такими поставщиками, как VMware, и в облаках, где бóльшая часть вычислений выполняется в каком-нибудь гипервизоре. Благодаря этому многие новички, управляющие инфраструктурой в облачно-ориентированном мире, могут даже не задумываться над вопросами, связанными с оборудованием. Структура, представленная на рис. 2.3, не является исчерпывающей иллюстрацией отличий между виртуальными и физическими серверами, а, скорее, показывает, чем они обычно отличаются с точки зрения взаимодействия.

Преимущества моделей виртуализации далеко не ограничиваются наличием единого API-интерфейса для взаимодействия. В виртуальных окружениях мы можем создать множество виртуальных серверов в рамках одного физического, что позволяет нам разделить каждый компьютер на полностью изолированные среды, в которых мы можем:

- легко создавать и клонировать серверы и их образы;
- использовать множество операционных систем на одном и том же сервере;

- оптимизировать функционирование сервера за счет выделения ресурсов в разных объемах в зависимости от потребностей приложения;
- изменять параметры ресурсов, не нарушая работу сервера;
- программным образом определять, имеют ли физические серверы доступ к таким ресурсам, как сетевые адаптеры;
- применять уникальные параметры сети и маршрутизации на каждом сервере.



Рис. 2.3. Объекты, с которыми взаимодействует администратор при выделении и управлении инфраструктурой на основе физического оборудования и виртуальных машин

Эта гибкость также позволяет нам выполнять работы по сопровождению в меньшем масштабе. Например, мы теперь можем обновить всего один хост, не влияя на все остальные, присутствующие на том же оборудовании. Кроме того, благодаря многим характеристикам, которые присущи виртуальным окружениям, создание и удаление серверов обычно становится более эффективным. Но у виртуализации есть свои недостатки. Как правило, чем сильнее вы абстрагируетесь от физического оборудования, тем выше ваши накладные расходы. Многие приложения с крайне высокими требованиями к производительности, такие как ПО для торговли ценными бумагами, лучше выполнять на физических серверах. Накладные расходы присущи и самим технологиям виртуализации. Для ресурсоемких вычислений (например, выполняемых телекоммуникационными компаниями в сетях 5G), предпочтительным может быть использование аппаратного обеспечения.

Итак, мы завершили краткий обзор революции, которая произошла в сфере виртуализации. Теперь давайте посмотрим, как она повлияла на использование Kubernetes и контейнерных абстракций, так как эти два фактора делают наше взаимодействие со стеком технологий еще более высокоуровневым. На рис. 2.4 показано, как это выглядит с точки зрения администратора, работающего со слоем Kubernetes. Компьютеры, лежащие в основе системы, воспринимаются как "море вычислений", в котором рабочие задания могут определять, какие ресурсы им нужны, после чего выделение этих ресурсов планируется соответствующим образом.

Необходимо отметить, что у кластеров Kubernetes есть несколько точек интеграции с внутренней инфраструктурой. Например, многие используют драйверы на основе CSI для поддержки разных хранилищ. Существует ряд проектов по управлению инфраструктурой, которые позволяют запрашивать у провайдера новые хосты и

присоединять их к кластеру. В организациях чаще всего применяются CPI (Cloud Provider Integrations), средства интеграции с облачными провайдерами, которые предлагают такие дополнительные возможности, как выделение балансировщиков нагрузки за пределами кластера для маршрутизации внутреннего трафика.



Рис. 2.4. Взаимодействие администратора с Kubernetes

В сущности, при отказе от виртуализации команда, отвечающая за инфраструктуру, теряет множество удобств — вещей, которые платформа Kubernetes сама по себе *не* предоставляет. Но существует несколько проектов и точек интеграции с аппаратным обеспечением, которые делают это направление еще перспективней. Ведущие облачные провайдеры начинают предлагать физические серверы, а сервисы IaaS наподобие Packet (недавно приобретенный компанией Equinix Metal, **https://metal.equinix.com**), которые работают без виртуальной прослойки, расширяют свое присутствие на рынке. Но успех в реализации физической инфраструктуры дается нелегко и сопряжен с такими проблемами:

- *Узлы существенно большего размера.* Как правило, чем больше узел, тем больше Pod'ов на нем размещается. Если для эффективной работы оборудования на одном узле необходимо разместить тысячи Pod-оболочек, это усложняет администрирование. Например, при выполнении обновлений на активном узле его необходимо опустошить, что может инициировать более 1000 событий повторного планирования.
- *Динамическое масштабирование.* Как быстро развертывать новые узлы с учетом нагрузки или трафика?
- *Выделение образов.* Быстрое создание и распространение системных образов для подержания как можно более высокой степени неизменяемости узлов кластера.
- *Нехватка API-интерфейса для балансировки нагрузки.* Необходимость в перенаправлении трафика, приходящего из-за пределов кластера, к внутренней сети Pod.
- *Менее развитая интеграция с хранилищами данных.* Организация доступа Pod'ов к сетевым хранилищам требует дополнительных усилий.

♦ *Вопросы безопасности, связанные с мультитенантным.* Когда используется виртуализация, мы можем позволить себе размещать контейнеры, требующие дополнительной безопасности, в отдельных гипервизорах. В частности, мы можем разделить физический сервер произвольным образом и организовать на основе этого планирование развертывания контейнеров.

Указанные проблемы, вне всякого сомнения, поддаются решению. Например, отсутствие интеграции с балансировщиком нагрузки можно решить с помощью таких проектов, как kube-vip (**https://kube-vip.io**) или metallb (**https://metallb.universe.tf**). Интеграцию с хранилищем можно обеспечить за счет кластера Ceph. Но суть состоит в том, что контейнеры не являются новаторской технологией виртуализации. Внутри большинства их реализаций применяются механизмы ядра Linux, которые делают так, чтобы процессы на одном и том же хосте "чувствовали" себя изолированными. Мы можем продолжать обсуждение этих нюансов до бесконечности, но, по сути, при выборе между облачными провайдерами (виртуализацией), локальной виртуализацией и физической инфраструктурой мы рекомендуем обращать внимание на то, какой из этих вариантов лучше всего отвечает вашим техническим требованиям и эксплуатационным процедурам, принятым в вашей организации. Если платформа Kubernetes рассматривается в качестве замены стеку виртуализации, еще раз подумайте о том, какие именно задачи она решает. Помните, что научиться эксплуатировать Kubernetes и обучить этому ваших работников, само по себе непросто. Тем более полное переосмысление того, как вы управляете своей внутренней инфраструктурой, существенно усложняет работу инженеров и повышает риски.

Выбор размера для кластера

Размер кластера, который вы собираетесь использовать, является неотъемлемой частью проектирования вашей модели развертывания Kubernetes и планирования инфраструктуры. Нас часто спрашивают, "сколько рабочих узлов должно быть в производственных кластерах?" Этот вопрос отличается от вопроса "сколько рабочих узлов необходимо для выполнения приложений?" Если вы планируете ограничиться одним-единственным кластером для всех своих задач, то ответ на оба вопроса будет один и тот же. Однако такое явление в "дикой природе" нам еще не встречалось. По аналогии с тем, как кластер Kubernetes позволяет относиться к серверным компьютерам, как к чему-то одноразовому, современные методы установки Kubernetes и облачные провайдеры позволяют применять такое же отношение и к самим кластерам. И на любом предприятии, в котором развернут Kubernetes, таких одноразовых кластеров как минимум несколько.

Преимущества более крупных кластеров:

♦ *Оптимизация затрат ресурсов.* Каждый кластер влечет за собой накладные расходы в виде плоскости управления, включая etcd и такие компоненты, как API-сервер. Кроме того, в каждом кластере выполняются разнообразные сервисы платформы, например, прокси-серверы на основе контроллеров Ingress. Все это делает накладные расходы еще больше. Крупный кластер минимизирует репликацию этих сервисов.

- *Менее частое развертывание кластеров.* Если вы используете собственную физическую инфраструктуру вместо локальной платформы виртуализации или услуг облачного провайдера, развертывание и удаление кластеров по необходимости, а также их масштабирование в соответствии с текущей нагрузкой становится более трудной задачей. Ваша стратегия развертывания кластеров может быть менее автоматизированной, если вы применяете ее не так часто. Вполне возможно, что трудозатраты на полную автоматизацию развертывания кластеров были бы выше трудозатрат, которые влечет за собой менее автоматизированная стратегия.
- *Упрощение подхода к администрированию кластера и рабочих заданий.* Если у вас не много промышленных кластеров, то системы, которые вы используете для их развертывания, объединения в федерацию и администрирования, могут быть менее элегантными и функциональными. Управление федеративными кластерами и приложениями является сложной задачей, требующей усилий. Сообщество работает над этой проблемой. Большие команды на крупных предприятиях активно разрабатывают системы для ее решения. Но пока что эти инициативы имеют ограниченный успех.

Преимущества кластеров меньших размеров:

- *Меньший масштаб потенциальных проблем.* Неполадки, возникающие в кластерах, будут затрагивать меньшее число приложений.
- *Гибкая арендность.* Kubernetes предоставляет все механизмы, необходимые для создания мультитенантной платформы. Однако в некоторых случаях развертывание нового кластера для отдельного арендатора требует намного меньше инженерных усилий. Например, если одному арендатору нужно воспользоваться общекластерным ресурсом, таким как CRD (Custom Resource Definition — пользовательское определение ресурса), а другой хочет иметь строгие гарантии изоляции в целях безопасности и/или соблюдения нормативно-правовых требований, то выделение каждому из них по отдельному кластеру может быть оправданным, особенно если рабочим заданиям необходимы существенные вычислительные ресурсы.
- *Меньше работы по адаптации к крупномасштабным задачам.* Когда число экземпляров приложений в кластере достигает нескольких сотен, мы часто сталкиваемся с проблемами масштабирования, которые необходимо решать. Эти проблемы варьируются от случая к случаю, но в вашей плоскости управления могут возникать узкие места. Инженерные ресурсы должны быть направлены на диагностику и адаптацию кластеров. В кластерах меньшего размера таких ресурсов требуется намного меньше.
- *Дополнительные способы обновления.* Уменьшение размера кластеров делает их обновление путем полной замены более доступным. Конечно, замена кластера имеет свои трудности, и о них мы поговорим позже, в разделе "Обновления" данной главы, но во многих случаях эта стратегия выглядит привлекательно, и использование меньших кластеров делает ее еще более заманчивой.

- *Альтернатива пулам узлов.* Если у вас есть приложения, которым нужны специальные ресурсы, такие как графические адаптеры или узлы с увеличенным объемом памяти, и ваши системы рассчитаны на работу с множеством мелких кластеров, эти особые требования будет легко удовлетворить путем выделения отдельных кластеров. В результате можно избавиться от трудностей, связанных с управлением разными пулами узлов, о чем мы уже упоминали ранее в этой главе.

Вычислительная инфраструктура

Кластеру Kubernetes нужны серверы — это само собой разумеется. В конце концов, его основное назначение состоит в управлении пулами этих серверов. О выборе подходящего оборудования следует позаботиться заранее. Сколько ядер вам необходимо? Сколько памяти? Сколько места на внутреннем накопителе? Какого класса сетевой интерфейс? Нужны ли вам какие-нибудь специализированные устройства наподобие графических адаптеров? Все эти вопросы продиктованы потребностями программного обеспечения, которое вы применяете. Чувствительны ли ваши приложения к вычислительным ресурсам? Требуют ли они много памяти? Применяете ли вы машинное обучение или ИИ, нуждающиеся в наличии GPU? Если у вас типичный сценарий использования в том смысле, что для ваших приложений хорошо подходит соотношение процессорной мощи и памяти, характерное для серверов общего назначения, и если эти приложения не слишком сильно отличаются по своему потреблению ресурсов, то эта задача будет относительно простой. Но, если вам требуется не совсем типичное и более разнообразное ПО, то придется уделить этому чуть больше внимания. Давайте рассмотрим разные виды серверов, которые можно выбрать для вашего кластера:

- *Серверы etcd (необязательно).* Этот тип серверов нужен только в случае, если вы используете для своих кластеров Kubernetes отдельные кластеры etcd. Плюсы и минусы такой конфигурации уже обсуждались в одном из предыдущих разделов. Приоритет в этих серверах должен отдаваться скорости чтения/записи, поэтому никогда не ставьте в них старые вращающиеся жесткие диски. Вы также можете предусмотреть отдельный диск для etcd, даже если ваши серверы выделенные, благодаря этому etcd не будет соперничать за дисковые ресурсы с ОС или другими программами. Еще следует подумать о сетевой производительности, в том числе о близости размещения в сети, чтобы снизить сетевые задержки между серверами в отдельно взятом кластере etcd.
- *Узлы плоскости управления (обязательно).* Эти серверы предназначены для работы компонентов плоскости управления кластера. Это должны быть компьютеры общего назначения, системные ресурсы которых соответствуют предполагаемому размеру кластера и требованиям к отказоустойчивости. В большом кластере API-сервер будет иметь больше клиентов и обрабатывать больший объем трафика. В этом случае можно выбрать серверы с большим количеством ресурсов или предусмотреть больше серверов. Однако такие компоненты, как планировщик и диспетчер контроллеров, имеют всего по одному активному ведущему серверу. Их емкость нельзя повысить за счет большего количества реплик, как в

случае с API-сервером. Если этим компонентам перестанет хватать ресурсов, вам нужно будет применить вертикальное масштабирование, делая каждый сервер более мощным. Кроме того, если на этих серверах вместе с плоскостью управления размещены агенты etcd, то также потребуется принять во внимание факторы, перечисленные в предыдущем пункте.

- *Рабочие узлы (обязательно).* Это серверы общего назначения, на которых выполняются приложения, не имеющие отношения к плоскости управления.
- *Узлы с повышенным объемом памяти (необязательно).* Если рабочие узлы общего назначения плохо подходят для ваших рабочих заданий ввиду того, что последние имеют повышенные требования к памяти, подумайте об использовании пула узлов, оптимизированного специально для этого ресурса. Например, если в качестве рабочих узлов выбран тип серверов AWS общего назначения M5 с соотношением вычислительных ресурсов и памяти вида 1CPU:4ГиБ, а ваши задачи потребляют ресурсы в соотношении 1CPU:8ГиБ, возникнет простой процессоров в вашем кластере. Для повышения эффективности можно использовать узлы с повышенным объемом памяти, такие как тип серверов AWS R5, которые имеют соотношение 1CPU:8ГиБ.
- *Узлы, оптимизированные для вычислений (необязательно).* Бывает, что приложениям подходят узлы, оптимизированные для вычислений, такие как типы серверов AWS C5 с соотношением 1CPU:2ГиБ. В таких случаях для повышения эффективности подумайте о добавлении пула узлов с такого рода серверами.
- *Узлы со специализированным оборудованием (необязательно).* Существует большой спрос на графические адаптеры. Если вашим приложениям (которые, например, занимаются машинным обучением) нужно специализированное оборудование, целесообразно добавить в кластер отдельный пул узлов с последующим размещением на них подходящих задач.

Сетевая инфраструктура

Работу с сетью можно воспринимать как один из аспектов реализации и легко ею пренебречь, но в результате могут возникнуть серьезные последствия для ваших моделей развертывания. Давайте для начала рассмотрим характеристики, которые вам нужно учитывать при проектировании.

Возможность маршрутизации

Вам практически наверняка не захочется делать узлы своего кластера публично доступными в Интернете. Удобство, которое дает возможность подключения к этим узлам откуда угодно, почти никогда не оправдывает потенциальные угрозы несанкционированного проникновения. Если вам нужно подключаться к этим узлам, то следует позаботиться о контроле доступа к ним. Но есть и более простое решение: хорошо защищенный узел-бастион или инсталляционный сервер с доступом по SSH.

Однако есть более тонкие вопросы, требующие ответа. Например, как организовать контроль доступа в вашей частной сети? У вас будут сервисы, которым требуется доступ к ресурсам как внутри, так и за пределами кластера. Например, нам часто необходима связь с массивами хранилищ, внутренними реестрами контейнеров, системами CI/CD, внутренними DNS-серверами, частными NTP-серверами и т. д. Вашему кластеру также, скорее всего, нужен будет доступ к таким публичным ресурсам, как общедоступные реестры контейнеров, даже если для этого применяется исходящий прокси-сервер.

Если доступ к Интернету исключен, то все ресурсы наподобие образов контейнеров и системных пакетов с открытым исходным кодом нужно будет сделать доступными каким-то другим образом. Отдавайте предпочтение более простым системам, для которых характерны согласованность и эффективность. По возможности избегайте необдуманных предписаний и ручного согласования инфраструктуры, необходимой для развертывания кластеров.

Резервирование

Используйте зоны доступности (англ. Availability Zones или AZ) для обеспечения непрерывной работы везде, где это возможно. Для ясности уточним, что зона доступности — это центр обработки данных с отдельным источником питания, системой резервного копирования и подключением к Интернету. Две подсети в одном ЦОД с общим источником питания нельзя считать двумя зонами доступности. Примером двух AZ могут служить два отдельных центра обработки данных в относительной близости друг к другу, которые имеют сетевое соединение с низкой сетевой задержкой и высокой пропускной способностью. Две зоны доступности — это хорошо, а три — еще лучше. Если говорить о большем количестве, то это зависит от масштабов бедствий, к которым вам нужно быть готовыми. Центры обработки данных уже неоднократно выходили из строя. Сбои сразу в нескольких ЦОД в одном и том же регионе возможны, но случаются редко и свидетельствуют о такой катастрофе, которая заставит вас задуматься о том, насколько важны ваши приложения. Для чего они предназначены: для национальной безопасности или для Интернет-магазина? Также подумайте о том, где именно требуется резервирование. Вы готовите резервные ресурсы для своих задач или плоскости управления самого кластера? Наш опыт показывает, что размещение etcd в разных AZ приемлемо, но при этом следует помнить о вопросах, поднятых в разделе "Сетевые условия" данной главы. Также имейте в виду, что распределение плоскости управления между разными зонами доступности создает резервные средства управления кластером. Выход плоскости управления из строя не повлияет на доступность ваших приложений, если только они от нее не зависят (чего следует избегать). Пострадает только ваша способность вносить какие-либо изменения в активное программное обеспечение. Неисправность плоскости управления — это серьезное и высокоприоритетное происшествие. Но это не то же самое, что сбой в приложениях, с которыми взаимодействуют пользователи.

Балансировка нагрузки

Вашим API-серверам нужен балансировщик нагрузки. Можете ли вы развернуть его программным образом в своем окружении? Если да, то его можно будет сконфигурировать в рамках развертывания плоскости управления вашего кластера. Продумайте политику доступа к API-интерфейсу и проанализируйте, за какими брандмауэрами будет находиться ваш балансировщик нагрузки. Его почти наверняка не стоит делать публично доступным из Интернета. Удаленный доступ к плоскости управления намного чаще осуществляется через сеть VPN, предоставляющую доступ к локальной сети, где находится ваш кластер. С другой стороны, если у вас есть публично доступные рабочие задания, вам понадобится отдельный балансировщик, через который проходит входящий трафик вашего кластера. В большинстве случаев этот балансировщик обслуживает все входящие запросы, направленные к различным приложениям. Развертывание отдельного балансировщика нагрузки и входящего соединения для каждого приложения, принимающего запросы снаружи, имеет сомнительную пользу. Если у вас есть отдельный кластер etcd, не размещайте балансировщик нагрузки между ним и Kubernetes API. Клиент etcd, которым пользуется API-интерфейс, может обрабатывать свои соединения без помощи балансировщика нагрузки.

Стратегии автоматизации

В ходе автоматизации инфраструктурных компонентов для своих кластеров Kubernetes вам придется принимать некоторые стратегические решения. Мы разделим их на две категории. Сначала речь пойдет об уже существующих инструментах, которыми вы можете пользоваться. Затем мы поговорим о том, как в этих целях можно задействовать операторы Kubernetes. Мы понимаем, что средства автоматизации для физической инфраструктуры выглядят совершенно иначе, поэтому давайте предположим, что у нас есть API-интерфейс, позволяющий выделять новые серверы и добавлять их в пул, доступный для развертывания Kubernetes. Если это не так, то вам придется восполнить пробелы вручную, чтобы в итоге обеспечить программный доступ и контроль. Начнем с инструментов, которые уже могут иметься в вашем распоряжении.

Средства управления инфраструктурой

Такие инструменты, как Terraform (**https://www.terraform.io**) и CloudFormation для AWS (**https://aws.amazon.com/cloudformation**), позволяют объявить состояние, в котором вы хотите видеть свою вычислительную и сетевую инфраструктуру, и затем применить его. Они используют форматы данных и языки конфигурации, с помощью которых вы можете описать нужный вам исход в виде текстовых файлов и затем попросить программный компонент воплотить в жизнь состояние, описанное в этих файлах.

Они полезны, так как задействуют инструментарий, который инженеры могут легко внедрить в своих системах и получить с его помощью желаемые результаты. Эти средства хорошо подходят для упрощения процесса развертывания относительно

сложной инфраструктуры. Они отлично себя показывают в условиях, когда у вас есть заранее определенный набор инфраструктурных компонентов, которые нужно создавать снова и снова, и когда их экземпляры не сильно варьируются. Это хорошо ложится на принцип неизменяемой инфраструктуры, так как ее повторяемость гарантируется, а полная *замена* компонентов вместо их *изменения* облегчает администрирование.

Ценность этих инструментов уменьшается по мере того, как требования к инфраструктуре становятся существенно более сложными, динамичными и зависимыми от изменчивых условий. Например, если вы проектируете системы развертывания Kubernetes с поддержкой разных облачных провайдеров, то эти инструменты окажутся громоздкими. Такие форматы, как JSON, и языки конфигурации, плохо подходят для описания условных выражений и циклических функций. Здесь во всей красе себя показывают языки программирования общего назначения.

Средства управления инфраструктурой повсеместно успешно применяются на стадиях разработки. Конечно, в некоторых компаниях они функционируют и в эксплуатационных условиях. Но со временем с ними становится сложно работать, и нередко они приводят к появлению технического долга, который впоследствии почти невозможно погасить. В связи с этим вам следует тщательно продумать использование готовых или разработку для этих целей собственных операторов Kubernetes.

Операторы Kubernetes

Если средства управления инфраструктурой накладывают ограничения, оправдывающие написание программного обеспечения с помощью языков программирования общего назначения, то какую форму должно принять это ПО? Вы можете написать веб-приложение или утилиту командной строки для администрирования инфраструктуры Kubernetes. Если вы обдумываете разработку собственного ПО для этих целей, обязательно обратите внимание на операторы Kubernetes.

В контексте Kubernetes операторы управляют системами с помощью пользовательских ресурсов и созданных вами контроллеров. Контроллеры применяют мощный и надежный метод управления состоянием. Когда вы создаете экземпляр ресурса Kubernetes, контроллер, отвечающий за этот тип ресурсов, оповещается API-сервером посредством механизма слежения и затем использует спецификацию ресурса в качестве инструкций для достижения желаемого состояния. Поэтому расширение API-интерфейса Kubernetes за счет новых типов ресурсов, представляющих разные аспекты инфраструктуры, и разработка операторов Kubernetes для управления состоянием этих инфраструктурных ресурсов является очень действенным подходом. Более подробно об операторах Kubernetes речь пойдет в *главе 11*.

Именно для этого был создан проект с открытым исходным кодом Cluster API. Это набор операторов Kubernetes, которые позволяют управлять инфраструктурой кластеров Kubernetes. И вы, безусловно, можете его применять в своих целях. На самом деле, прежде чем самому браться за разработку чего-то подобного с нуля, мы советуем вам ознакомиться с этим проектом и посмотреть, соответствует ли он ва-

шим потребностям. И, если он вам не подходит, то уясните, может ли ваша команда внести свой вклад в Cluster API, чтобы разработать новые возможности и/или добавить поддержку других провайдеров инфраструктуры, которые вам нужны?

Kubernetes предлагает отличные средства автоматизации и администрирования контейнерных развертываний программного обеспечении, а также довольно выгодные стратегии автоматизации инфраструктуры кластеров посредством операторов Kubernetes. Как следует подумайте над возможностью их использования и, если это возможно, над участием в таких проектах, как Cluster API. Если у вас есть нестандартные требования, и вы предпочитаете управлять инфраструктурой с помощью инструментов, то их успешное применение вполне возможно. Однако ввиду ограниченности языков и форматов конфигурации по сравнению с полноценными языками программирования вашим решениям будет недоставать гибкости и изящности.

Развертывание серверов

Для серверов, которые вы запускаете для своего кластера, нужно подготовить операционную систему, установить определенные пакеты и написать конфигурацию. Вам также нужна какая-нибудь утилита или программа, которая будет определять переменные окружения и другие значения, применять их и координировать процесс запуска контейнерных компонентов Kubernetes.

Для этого существуют две стратегии общего характера:

- средства управления конфигурацией;
- системные образы.

Управление конфигурацией

Средства управления конфигурацией, такие как Ansible, Chef, Puppet и Salt, набирают популярность в мире, где программное обеспечение традиционно устанавливалось на виртуальные машины или прямо на физические компьютеры. Эти инструменты отлично подходят для автоматизации конфигурирования множества удаленных серверов. Они основаны на разных моделях, но в целом позволяют администраторам декларативно описывать характеристики сервера и автоматически применять это описание.

Несомненное преимущество этих средств управления конфигурацией — возможность достижения надежной согласованности серверов. Каждый сервер получает фактически идентичный набор программного обеспечения и конфигурации, что обычно реализуется с помощью декларативных рецептов или наборов предписаний, хранящихся в системе контроля версий. Все это делает их хорошим решением.

Однако в мире Kubernetes их недостатком является относительно невысокие скорость и надежность развертывания кластеров. Если процесс, с помощью которого вы присоединяете к кластеру новый рабочий узел, включает в себя установку пакетов, которые требуют скачивания ресурсов по сети, этот узел кластера входит в строй с существенной задержкой. Более того, во время конфигурации и установки

случаются ошибки. Процесс управления конфигурацией может провалиться по целому ряду причин, от временно недоступных репозиториев с пакетами до некорректных переменных, что делает присоединение узла к кластеру невозможным. И, если ваш кластер испытывает нехватку ресурсов и полагается на автоматическое масштабирование, то неудачное присоединение подобного узла может спровоцировать или продлить проблемы с доступностью.

Системные образы

Более предпочтительная альтернатива — использование системных образов. Если в вашем образе уже установлены все необходимые пакеты, программное обеспечение будет готово к работе сразу после загрузки сервера. Мы избавляемся от процесса установки, которому нужны доступ к сети и доступные репозитории пакетов. Системные образы делают присоединение узлов к кластеру более надежным и позволяют намного быстрее подготовить их к приему трафика.

Изящности этому методу придает тот факт, что для создания системных образов во многих случаях подойдут уже знакомые вам средства управления конфигурацией. Например, Packer (**https://www.packer.io**) от HashiCorp позволяет применять Ansible для построения образов в формате AMI (Amazon Machine Image), которые затем при необходимости можно развертывать на ваших серверах. Если при создании системного образа с помощью плейбука Ansible возникнет ошибка, в этом не будет ничего страшного. А вот если та же ошибка не даст рабочему узлу присоединиться к кластеру, это может привести к серьезному происшествию в условиях эксплуатации.

Ресурсы, необходимые для сборки образов, можно (и следует) хранить в системе контроля версий. Таким образом все аспекты процессов установки будут представлены в декларативном виде и доступны всем, кто просматривает репозиторий. Если нужно перейти на новую версию или применить исправления безопасности, то эти ресурсы после слияния всегда можно обновить, зафиксировать и, в идеале, автоматически применить в рамках конвейера CI/CD.

Одни решения требуют тщательного взвешивания, а другие совершенно очевидны. Использование заранее подготовленных системных образов относится к последним.

Что устанавливать

Так что же нужно установить на сервере?

Начнем с очевидного: вам понадобится операционная система. Удачным выбором будет дистрибутив Linux, в сочетании с которым многие используют и тестируют Kubernetes. В этой связи можно посоветовать RHEL/CentOS или Ubuntu. Если вы предпочитаете другой дистрибутив (или есть его коммерческая поддержка), и готовы потратить чуть больше времени на тестирование и разработку, то в этом тоже нет ничего плохого. Будет еще лучше, если вы выберете дистрибутив, специально созданный для контейнеров, такой, как Flatcar Container Linux (**https://www.flatcar-linux.org**).

Следующим в списке очевидных компонентов идет среда выполнения контейнеров, такая, как Docker или containerd. Если у вас есть контейнеры, то нужна среда для их выполнения.

Дальше можно упомянуть kubelet. Это интерфейс между платформой Kubernetes и контейнерами, которые она оркестрирует. Он устанавливается на сервер, координирующий работу контейнеров. Kubernetes имеет контейнерную экосистему. На сегодня саму платформу Kubernetes принято использовать внутри контейнеров. Тем не менее, kubelet — это один из тех компонентов, которые выполняются прямо на хосте в виде обычных исполняемых файлов или процессов. Попытки применить его внутри контейнера уже предпринимались, но это только все усложняло. Не делайте этого. Устанавливайте kubelet на хост, а остальные компоненты Kubernetes выполняйте в контейнерах. Это четкий и практичный подход.

Итак, мы имеем ОС Linux, среду для выполнения контейнеров и интерфейс между Kubernetes и этой средой. Теперь нам нужно каким-то образом подготовить начальную конфигурацию плоскости управления Kubernetes. Утилита kubelet умеет запускать контейнеры, но без плоскости управления она "не знает", какие Pod'ы нужно развернуть. Здесь нам понадобятся kubeadm и статические Pod'ы.

Kubeadm — это далеко не единственный инструмент, способный выполнить такую начальную конфигурацию. Но он получил широкое распространение в сообществе и успешно применяется в системах многих предприятий. Это программа командной строки, в обязанности которой среди прочего входит генерация статических Pod-манифестов, необходимых для настройки и работы Kubernetes. Утилиту kubelet можно сконфигурировать так, чтобы она отслеживала директорию на хосте и запускала Pod'ы для любых Pod-манифестов, которые там будут появляться. Kubeadm применит к kubelet именно такую конфигурацию и при необходимости сохранит манифесты в нужное место, подготовив основные компоненты плоскости управления Kubernetes — в частности, etcd, kube-apiserver, kube-scheduler и kube-controller-manager.

Впоследствии kubelet будет получать все дальнейшие инструкции по созданию Pod-оболочек на основе манифестов, передаваемых API-интерфейсу Kubernetes. Кроме того, kubeadm сгенерирует токены начальной конфигурации, с помощью которых вы сможете безопасно добавлять другие узлы в свой новенький кластер.

Напоследок вам понадобится какая-нибудь *утилита начальной конфигурации*. Проект Cluster API задействует для этого пользовательские ресурсы и контроллеры Kubernetes. Но консольная программа, установленная на хосте, тоже подойдет. Основная функция этой утилиты состоит в том, чтобы вызывать команду `kubeadm` и управлять конфигурациями сред выполнения. Когда сервер загружается, аргументы, предоставленные этой утилите, позволяют ей выполнить начальную конфигурацию Kubernetes. Например, в AWS утилите можно передать аргументы на основе пользовательских данных, которые проинформируют ее о том, какие флаги нужно добавить к команде `kubeadm` или какие параметры нужно включить в ее конфигурационный файл. Как минимум, речь идет о параметрах, по которым утилита начальной конфигурации сможет определить, что ей делать: создать новый кластер с по-

мощью команды `kubeadm init` или присоединить сервер к существующему кластеру командой `kubeadm join`. Также следует указать безопасное место для хранения токена начальной конфигурации (при инициализации) или его извлечения (в случае присоединения). Эти токены гарантируют, что к вашему кластеру могут присоединиться только одобренные серверы, поэтому относитесь к ним бережно. Чтобы получить четкое представление о том, какую конфигурацию среды выполнения нужно предоставить в вашем случае, выполните ручную установку Kubernetes с использованием команды `kubeadm`, которая хорошо описана в официальной документации. Следуя этим инструкциям, вы увидите, что необходимо для удовлетворения вашим требованиям в вашем окружении. На рис. 2.5 проиллюстрированы этапы развертывания нового сервера для создания первого узла плоскости управления в кластере Kubernetes.

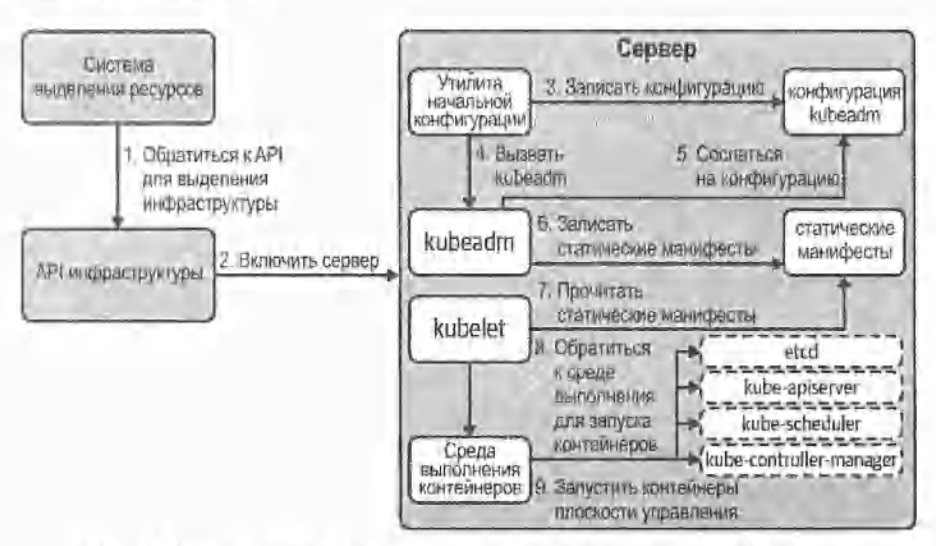


Рис. 2.5. Начальная конфигурация сервера для инициализации Kubernetes

Итак, мы обсудили, что следует устанавливать на серверы, которые являются частью кластера Kubernetes. Теперь перейдем к программному обеспечению, которое выполняется в контейнерах и составляет плоскость управления Kubernetes.

Контейнерные компоненты

Статические манифесты, предназначенные для развертывания кластера, должны содержать основные компоненты плоскости управления: etcd, kube-apiserver, kube-scheduler и kube-controllermanager. При необходимости можно предоставить дополнительные пользовательские Pod-манифесты, но исключительно для тех Pod'ов, которые обязательно должны быть запущены еще до того, как станет доступным или будет зарегистрирован в федеративной системе API-интерфейс Kubernetes. Если рабочее задание можно установить позже, с помощью API-сервера, так и следует сделать. Любыми Pod'ами, созданными на основе статических манифестов, можно

управлять только путем редактирования этих манифестов на диске сервера, что куда менее удобно и хуже автоматизируется.

Если вы используете kubeadm, что настоятельно рекомендуется, при инициализации узла плоскости управления с помощью команды `kubeadm init` будут созданы соответствующие статические манифесты, в том числе и для etcd. Любые флаги, которые вы хотите указать для этих компонентов, можно передать kubeadm посредством конфигурационного файла. Например, утилита начальной конфигурации, которая вызывает kubeadm (мы обсуждали ее в предыдущем разделе), может записать этот конфигурационный файл на основе шаблона.

Статические Pod-манифесты лучше не редактировать вручную с помощью утилиты начальной конфигурации. Если это действительно необходимо, этапы создания статического манифеста и инициализации кластера можно выполнить отдельно, используя kubeadm; это позволит внедрить нужные вам изменения. Однако так следует делать только для важных изменений, которые невозможно внести посредством конфигурации kubeadm. Чем проще процесс начальной конфигурации плоскости управления Kubernetes, тем он будет надежней, быстрее и менее подвержен сбоям при переходе на новую версию.

Kubeadm также сгенерирует самозаверенные TLS-сертификаты, необходимые для безопасного подключения компонентов вашей плоскости управления. Опять же, здесь лучше ничего не менять. Если требования безопасности предусматривают использование в качестве источника доверия вашего корпоративного центра сертификации, это можно организовать. В таком случае важно иметь возможность автоматизировать получение одобрения со стороны промежуточного центра сертификации. И помните, что безопасность ваших механизмов начальной конфигурации кластера гарантирует подлинность самозаверенных сертификатов, и вы можете быть уверены в том, что они будут действительны только для плоскости управления отдельно взятого кластера.

Мы обсудили все тонкости установки Kubernetes. Теперь давайте рассмотрим насущные проблемы, возникающие при использовании кластера. Начнем с установки дополнений, без которых нельзя обойтись. Это компоненты, которые позволят превратить Kubernetes в платформу приложений, пригодную для промышленного применения. Затем речь пойдет об аспектах и стратегиях обновления вашей платформы.

Дополнения

В этом разделе будут рассмотрены дополнительные сервисы платформы, которые работают поверх кластера Kubernetes. Мы не станем давать советы о том, *какие* дополнения нужно устанавливать. Этому, в сущности, посвящены оставшиеся главы. Здесь же мы поговорим о том, *как* установить компоненты, которые превратят ваш стандартный кластер Kubernetes в платформу промышленного уровня, которой будет удобно пользоваться разработчикам.

Дополнения, которые устанавливаются в кластер, следует считать частью модели развертывания. Их установка обычно является заключительным этапом разверты-

вания кластера. Их администрирование и версионирование должно проводиться в сочетании с самим кластером Kubernetes. К дополнениям, составляющим платформу, и Kubernetes удобно относиться как к пакету, который тестируется и выпускается как единое целое, так как между определенными компонентами платформы неизбежно возникают версионные и конфигурационные зависимости.

Kubeadm устанавливает "обязательные" дополнения, необходимые для прохождения проверок на соответствие проекту Kubernetes, включая DNS-сервер кластера и компонент kube-proxy, который реализует ресурсы Service. Однако существует множество других, не менее важных компонентов, которые нужно будет добавить после того, как kubeadm завершит свою работу. Самый яркий пример — CNI (Container Networking Interface — интерфейс управления сетью контейнеров). Без сети Pod-оболочек ваш кластер будет мало на что способен. Достаточно сказать, что для расширения возможностей платформы, которую вы создаете поверх Kubernetes, вам придется добавить в свой кластер довольно много компонентов, обычно в виде таких ресурсов, как DaemonSet, Deployment или StatefulSet.

Ранее в этой главе в разделе "Архитектура и топология" мы обсуждали федерацию и добавление в нее новых кластеров. Это обычно делается перед установкой дополнений, так как системы и определения для процесса установки зачастую размещаются в управляющем кластере.

Какой бы ни была архитектура, после завершения регистрации кластера можно инициировать установку дополнений. Этот процесс будет состоять из последовательных обращений к API-серверу с просьбой создать ресурсы Kubernetes, необходимые каждому компоненту. Эти обращения могут поступать как изнутри кластера, так и от систем, которые находятся за его пределами.

Один из подходов к установке дополнений состоит в использовании конвейера непрерывной доставки с применением готовых инструментов вроде Jenkins. В данном контексте "непрерывность" не играет никакой роли, так как этот процесс инициируется не обновлением ПО, а скорее появлением новой цели для установки. В CI и CD "непрерывность" обычно означает автоматическое выкатывание программного обеспечения при интеграции изменений в определенную ветвь исходного кода в системе контроля версий. Инициирование установки дополнений в недавно развернутом кластере — это совершенно другой механизм, но конвейер CI/CD здесь может пригодиться, так как он обычно поддерживает функции, необходимые для процесса установки. Вам остается только реализовать вызов для запуска конвейера в ответ на создание нового кластера, а также предоставить любые переменные, которые позволят провести установку корректно.

Еще один, более традиционный для Kubernetes подход заключается в использовании оператора Kubernetes. Вы должны расширить API-интерфейс за счет добавления одного или нескольких пользовательских ресурсов, которые позволят вам определить дополнительные компоненты кластера и их версии. Этот более сложный способ также подразумевает написание логики контролера, способной выполнить корректную установку дополнений на основе спецификации, определенной в пользовательском ресурсе.

Польза этого подхода состоит в том, что он предоставляет четкий централизованный источник информации о дополнениях, установленных в кластере. Но, что более важно, он дает возможность управления жизненным циклом этих дополнений программным путем. Его недостаток состоит в необходимости разрабатывать и поддерживать более сложное программное обеспечение. Если вы на это готовы, то нужно обеспечить возможность последующих обновлений этих дополнительных компонентов и их дальнейшего администрирования, что избавит вас от существенных усилий. Если ограничиться одной лишь установкой и не разработать механизмы для выполнения обновлений, это ляжет тяжелым бременем на плечи разработчиков ПО без какой-либо существенной выгоды. Операторы Kubernetes приносят больше всего пользы при проведении постоянных эксплуатационных процедур, позволяя отслеживать пользовательские ресурсы, которые обеспечивают желаемое состояние.

Следует прояснить один момент: идею оператора дополнений не обязательно реализовывать отдельно от внешних систем, таких как CI/CD. На самом деле они с большой вероятностью будут функционировать в связке. Например, вы можете установить пользовательские ресурсы оператора и дополнения с помощью конвейера CD и затем переложить дальнейшую работу на сам оператор. К тому же, оператору, скорее всего, нужно будет получить манифесты для установки, которые могут находиться в том же репозитории кода, что и шаблонные манифесты Kubernetes для дополнений.

Использование оператора в такой манере сокращает внешние зависимости, что способствует повышению надежности. Однако от внешних зависимостей нельзя полностью избавиться. Управление дополнениями с помощью оператора Kubernetes следует предпринимать, только если ваши инженеры знакомы с этим шаблоном проектирования и имеют опыт его применения. В противном случае лучше ограничиться инструментами и системами, которыми ваша команда уже хорошо владеет, и продолжить расширение познаний и опыта своих коллег в этой предметной области.

На этом мы заканчиваем обсуждение вопросов, касающихся систем для установки кластера Kubernetes и его дополнений. Теперь пришло время поговорить об обновлениях.

Обновления

Управление жизненным циклом кластера имеет непосредственное отношение к его развертыванию. Система развертывания кластера может и не быть рассчитана на будущие обновления, однако их поддержку рекомендуется предусмотреть, так как эти два процесса имеют достаточно много общего. Как минимум, прежде чем развертывать свое ПО в промышленном окружении, необходимо позаботиться о стратегии его обновления. Развертывание платформы без возможности ее обновлять и обслуживать в лучшем случае является рискованным. Если вы видите рабочие задания, выполняющиеся на версии Kubernetes, которая существенно отстает от последнего выпуска, это результат разработки системы развертывания кластера, ко-

торая была внедрена еще до реализации механизма обновлений. Когда вы впервые развертываете в промышленном окружении рабочие задания, генерирующие прибыль, существенная часть инженерных ресурсов обычно тратится на добавление недостающих возможностей и устранение острых проблем, с которыми сталкивается ваша команда. Со временем нужные возможности будут реализованы, а проблемы исправлены, но суть в том, что, пока вы этим занимаетесь, стратегия обновления ждет своей очереди. Выделяйте на нее ресурсы с самого начала. Впоследствии вы будете рады, что приняли это решение.

Обсуждение темы обновлений мы начнем с версионирования, которое позволит четко определить зависимости, как для самой платформы, так и для приложений, которые будут ее использовать. Мы также поговорим о том, как спланировать откат обновлений на случай, если что-то пойдет не так, и удостовериться в том, что все прошло по плану. В конце мы сравним и противопоставим конкретные стратегии обновления Kubernetes.

Версионирование платформы

Прежде всего, версионируйте свою платформу и документируйте версии всех ее компонентов. Это касается как версий операционных систем на серверах, так и всех пакетов, которые на них установлены, включая среду выполнения контейнеров и, конечно же, Kubernetes. Также имеют значение версии каждого дополнения к вашей платформе приложений. Довольно часто версия платформы совпадает с версией Kubernetes. Благодаря этому можно сразу сказать, что выпуск платформы 1.18 основан на Kubernetes 1.18, и для этого не нужно ни с чем сверяться. Но это не так уж важно по сравнению с самим фактом ведения и документирования версий. Работайте с любой системой, которую предпочитает ваша команда. Главное, чтобы эта система существовала, была задокументирована и неотступно соблюдалась. Единственное, что нам не нравится в привязке версии платформы к какому-либо ее компоненту, — это потенциальная путаница. Например, если вам нужно обновить вашу среду выполнения контейнеров из-за уязвимости безопасности, это должно быть отражено в версии вашей платформы. Если следовать принципам семантического версионирования, это, наверное, будет выглядеть как изменение номера патч-версии. Это можно спутать с изменением версии самого проекта Kubernetes — например, v1.18.5 → 1.18.6. Рассмотрите возможность назначения своей платформе отдельных номеров версий, особенно если вы используете семантические версии вида мажорная/минорная/патч. Подход, согласно которому программное обеспечение имеет отдельную версию и зависит от других компонентов с их собственными версиями, является чуть ли не универсальным. Если ваша платформа следует тому же принципу, то нумерация ее версий будет понятна всем инженерам без лишних слов.

Планирование на случай сбоев

Всегда исходите из того, что во время процесса обновления что-то может пойти не так. Представьте, что вам нужно восстановить систему после катастрофического

сбоя, это должно вызвать у вас чувство тревоги. Пусть оно будет вашей мотивацией для тщательной подготовки к такому исходу. Автоматизируйте процесс резервного копирования своих ресурсов Kubernetes и их восстановления — как в виде непосредственных снимков etcd, так и с помощью резервных копий Velero, создаваемых через API-интерфейс. Предусмотрите те же механизмы для постоянных данных, которые нужны вашим приложениям. Отдельно позаботьтесь о восстановлении после серьезных неполадок своих важнейших приложений и их зависимостей. Если речь идет о сложных, распределенных приложениях, хранящих свое состояние, то вам, скорее всего, придется учитывать порядок восстановления и взаимные зависимости компонентов. Предусмотрите все возможные режимы сбоев и разработайте системы автоматического восстановления с последующим их тестированием. Для самых важных рабочих заданий и их зависимостей подготовьте резервные кластеры, готовые взять на себя нагрузку, и по возможности займитесь их автоматизацией и тестированием.

Тщательно продумайте пути отката обновлений. Если новая версия сопровождается ошибками или неполадками, которые вы не можете сразу диагностировать, то возможность ее откатить будет хорошей страховкой. Диагностика сложных распределенных систем может занять некоторое время, а стресс, связанный с неполадками в промышленных развертываниях, может все усугубить. Заранее подготовленные процедуры и средства автоматизации, на которые можно положиться в таких ситуациях, будут как никогда важны, если речь идет о сложных платформах на основе Kubernetes. Но будьте практичны и реалистично смотрите на вещи. В реальности откатывать обновления не всегда разумно. Например, если процесс обновления уже зашел достаточно далеко, то откат всех внесенных изменений будет ужасной идеей. Подумайте об этом заранее, определите точки невозврата и выработайте подходящие стратегии, прежде чем выполнять эти операции в активной системе.

Интеграционное тестирование

У вас может быть хорошо документированная система версионирования, которая содержит информацию о версиях всех компонентов, однако управление этими версиями — совсем другая история. В таких сложных системах, как платформы на основе Kubernetes, довольно непросто добиться того, чтобы все всегда как следует друг с другом интегрировалось и работало вместе. Необходимо позаботиться о совместимости не только всех компонентов, но и рабочих заданий, которые выполняются на платформе. К тому же, сама платформа должна быть протестирована с подтверждением работоспособности. Старайтесь делать так, чтобы ваши приложения не зависели от какой-то конкретной платформы, чтобы уменьшить количество возможных проблем с совместимостью, хотя во многих случаях использование уникальных функций платформы приносит колоссальную пользу.

Важную роль играет модульное тестирование всех компонентов платформы и другие полезные методы разработки программного обеспечения. Однако не менее важны интеграционные тесты, несмотря на их куда более высокую сложность. Помочь в этом может Sonobuoy, отличная утилита тестирования на соответствие.

Чаще всего с ее помощью выполняют сквозные тесты в основной ветке разработки Kubernetes, чтобы гарантировать получение корректно работающего кластера, все компоненты которого работают вместе так, как ожидалось. Сканирование Sonobuoy нередко проводят после выделения нового кластера, чтобы автоматизировать операции, которые обычно выполняются вручную, такие как проверка Pod-оболочек плоскости управления и развертывание тестовых рабочих заданий для подтверждения работоспособности кластера. Но мы рекомендуем на этом не останавливаться. Напишите свои собственные подключаемые модули, которые проверяют конкретные функции и возможности вашей платформы. Тестируйте операции, от которых зависит удовлетворение требованиям вашей компании. И проводите такое сканирование регулярно. Используйте Kubernetes CronJob, чтобы выполнять если не полный набор тестов, то какую-то часть подключаемых модулей. На сегодня эти возможности изначально не доступны, но их можно легко реализовать, и это, несомненно, будет стоить потраченных усилий: предоставьте результаты сканирования в виде метрик, которые можно выводить на информационной панели и использовать для создания оповещений. Эти проверки на соответствие фактически позволяют убедиться в том, что разные части распределенной системы работают вместе, предоставляя функциональность и возможности, которые вы ожидаете. Это очень эффективный метод автоматического интеграционного тестирования.

Опять же, интеграционное тестирование должно охватывать и приложения, которые выполняются в рамках платформы. Авторы платформы должны активно это поощрять, хотя разные команды разработчиков неизбежно будут использовать для этого различные стратегии. Проводите интеграционное тестирование кластера, максимально приближенного к промышленному окружению (подробней об этом чуть позже). Это особенно важно для приложений, которые задействуют возможности платформы. Хороший пример — операторы Kubernetes, которые расширяют API-интерфейс Kubernetes и по своей природе глубоко интегрированы в платформу. Если вы применяете оператор для развертывания и администрирования жизненного цикла любой программной системы в своей организации, то интеграционное тестирование обязательно должно проводиться для разных версий вашей платформы, особенно при обновлении Kubernetes.

Стратегии

Мы рассмотрим три стратегии обновления платформ на основе Kubernetes:

- замена кластера;
- замена узлов;
- обновление без прекращения работы.

Они перечислены в порядке от наивысших затрат и наименьшего риска до самых низких затрат и наибольшего риска. Как это обычно бывает, идеального решения, которое бы подходило во всех ситуациях, не существует. Необходимо сбалансировать издержки и выгоды, чтобы найти решение, соответствующее вашим требованиям, бюджету и допустимым рискам. Более того, каждая стратегия может иметь разные степени автоматизации и тестирования, зависящие от таких факторов, как

средства, выделенные на инженерную работу, допустимость рисков и частота обновлений.

Помните, что эти стратегии не исключают друг друга. Вы можете их сочетать. Например, вы могли бы обновить выделенный кластер без прекращения работы и затем выполнить замену узлов для остальных компонентов кластера Kubernetes. Вы также *можете* применять различные стратегии на разных уровнях в зависимости от допустимых рисков. Однако рекомендуемый подход состоит в выборе везде одной и той же стратегии, чтобы методы, применяемые в промышленных условиях, были тщательно проверены на стадиях разработки и финального тестирования.

Какую бы стратегию вы ни выбрали, несколько принципов остаются неизменными: тщательно все тестируйте и обеспечивайте степень автоматизации, разумную с практической точки зрения. Если вы создадите средства автоматизации и тщательно их проверите в ходе тестирования и разработки кластеров, это существенно снизит вероятность того, что ваши обновления в промышленном окружении вызовут какие-либо проблемы для конечных пользователей и заставят напряженно работать вашу команду сопровождения платформы.

Замена кластера

Замена кластера — это самое затратное, но наименее рискованное решение. Невысокая степень риска объясняется тем, что принцип неизменяемой инфраструктуры применяется ко всему кластеру. Обновление состоит в развертывании совершенно нового кластера параллельно со старым. Приложения переносятся из старого в новый. При этом обновленный кластер по необходимости масштабируется. Число рабочих узлов в старом кластере уменьшается по мере переноса рабочих заданий. Однако на протяжении всего процесса обновления у вас будет совершенно отдельный кластер со всеми сопутствующими расходами. Эти расходы смягчаются за счет одновременного расширения нового кластера и уменьшения старого, т. е. при обновлении промышленного кластера с 300 узлами вам не нужно дополнительно выделять 300 узлов. Вы создадите кластер, например, с 20 узлами. После переноса первых нескольких приложений старый кластер можно уменьшить, так как нагрузка на него снизилась, а новый — расширить, чтобы подготовить место для дальнейшей миграции.

Автомасштабирование и создание дополнительного кластера может сделать этот процесс довольно простым, но одни лишь обновления вряд ли оправдают использование таких технологий. Существуют две распространенные проблемы, возникающие при замене кластера.

Первая — это управление входящим трафиком. Пока приложения переносятся из одного кластера в другой, соответствующий трафик необходимо перенаправлять к новому, обновленному кластеру. Это означает, что DNS-сервер для публично доступных приложений должен ссылаться не на входящий канал кластера, а на глобальный балансировщик нагрузки (англ. Global Service Load Balancer или GSLB) или обратный прокси, а тот уже направляет трафик в соответствующий канал. Это место, где вы можете управлять маршрутизацией трафика для разных кластеров.

Вторая трудность связана с доступностью постоянного хранилища. Если вы задействуете для хранения сервисы или специальные устройства, они должны быть доступны из обоих кластеров. Если вы пользуетесь управляемым сервисом баз данных от своего общедоступного облачного провайдера, то нужно позаботиться о том, чтобы к нему могли обращаться оба кластера. В закрытом центре обработки данных это может сводиться к организации сети и настройке брандмауэра. В общедоступном облаке это может быть вопрос все той же организации сети и зон доступности, например, тома AWS EBS можно использовать из определенных AZ. А у управляемых сервисов в AWS зачастую предусмотрены отдельные виртуальные облака (англ. Virtual Private Clouds или VPC). Поэтому вам следует подумать об использовании VPC для нескольких кластеров. Kubernetes нередко устанавливается так, что каждому кластеру выделяется по VPC, но эта модель не всегда оптимальна.

Дальше вам нужно позаботиться о переносе приложений. Речь в основном идет о самих ресурсах Kubernetes, таких как Deployment, Service, ConfigMap и т. д. Их можно переносить одним из двух способов:

1. Развернуть заново из объявленного источника информации.
2. Скопировать имеющиеся ресурсы из старого кластера.

Первый вариант, скорее всего, потребует переориентации вашего конвейера развертывания на новый кластер и повторное развертывание в этом кластере тех же ресурсов. Это подразумевает наличие надежного источника информации с определениями ваших ресурсов, хранящимися в системе контроля версий, отсутствие в ходе этого процесса каких-либо дополнительных изменений.

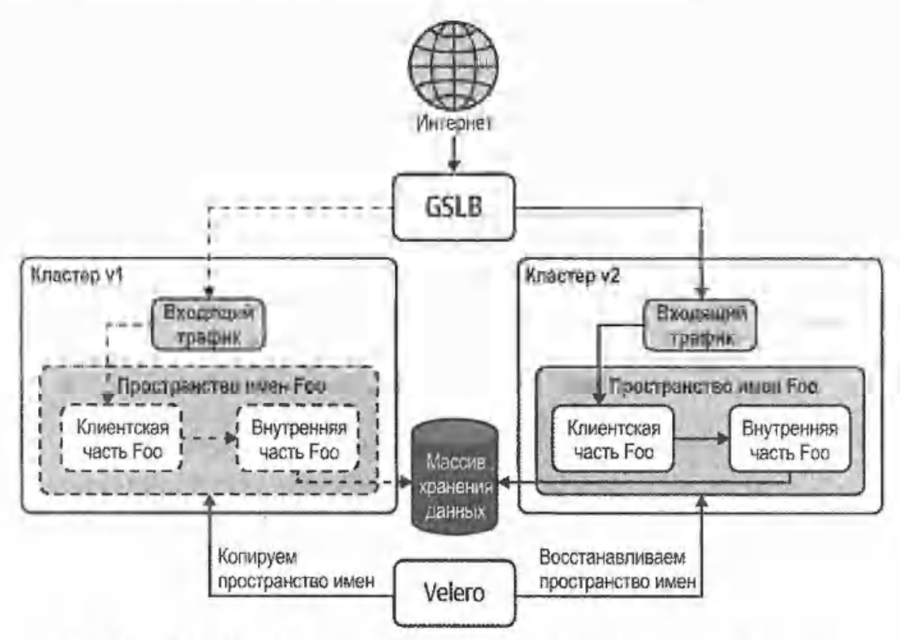


Рис. 2.6. Перенос рабочих заданий между кластерами путем их копирования и восстановления с помощью Velero

В реальности так бывает довольно редко. Обычно люди, контроллеры и другие системы успевают внести свои изменения и корректировки. В таких случаях вам придется выбрать второй вариант: скопировать имеющиеся ресурсы и развернуть их в новом кластере. Чрезвычайно полезны в этом процессе такие инструменты, как Velero. Velero чаще всего называют средством резервного копирования, но возможности миграции, которые он предоставляет, могут быть не менее, а то и более ценными. Velero может сделать снимок всех ресурсов в вашем кластере или их подмножества. Поэтому, если вы переносите приложения по одному пространству имен за раз, снимки пространств можно создавать в момент переноса, после чего у вас будет возможность очень надежным образом восстановить их уже в новом кластере. Эти снимки берутся не напрямую из хранилища данных etcd, а посредством API-интерфейса Kubernetes, и если предоставить Velero доступ к API-серверам обоих кластеров, то данный метод может оказаться весьма полезным. Это проиллюстрировано на рис. 2.6.

Замена узлов

Вариант с заменой узлов — это компромиссное решение с точки зрения расходов и рисков. Оно повсеместно распространено и имеет поддержку со стороны Cluster API. Его стоит применять для крупных кластеров, но при условии, что вы хорошо разбираетесь в вопросах совместимости. Эти вопросы являются одним из самых серьезных рисков данного метода, так как для ваших сервисов и приложений все выглядит так, будто плоскость управления продолжает работать во время обновления. Если в ходе этого процесса обнаружится, что версия API-интерфейса, которую использует одно из ваших рабочих заданий, больше не доступна, это задание может выйти из строя. Описанную проблему можно смягчить несколькими способами:

- Ознакомьтесь с примечаниями к выпуску Kubernetes. Прежде чем выкатывать новую версию платформы с обновленной версией Kubernetes, внимательно прочитайте журнал изменений (файл changelog). Там задокументированы все части API-интерфейса, которые были удалены или признаны устаревшими, поэтому у вас будет вдоволь времени, чтобы на них среагировать.
- Перед развертыванием в промышленном окружении проводите тщательное тестирование. Новые версии вашей платформы должны быть как следует проверены в кластерах, предназначенных для разработки и финального тестирования. Устанавливайте новые версии Kubernetes в среде для разработки сразу после их выхода. Это позволит вам их тщательно проверять и использовать в своем промышленном окружении всегда свежие выпуски Kubernetes.
- Старайтесь не слишком полагаться на API-интерфейс. Это не касается сервисов платформы, которые работают в вашем кластере, им по своей природе нужно тесно интегрироваться с Kubernetes. Однако приложения, которые выполняются в промышленных условиях и взаимодействуют с конечными пользователями, должны как можно меньше зависеть от конкретной платформы. API-интерфейс Kubernetes не должен выступать в роли зависимости. Например, вашему приложению не должно быть известно об объектах `Secret`. Оно обязано просто ис-

пользовать переменную окружения или читать файл, который ей предоставлен. Таким образом, если вы будете вовремя обновлять манифесты, с помощью которых развертывается ваше приложение, то его рабочее задание само по себе будет спокойно работать, не обращая внимания ни на какие изменения в API. Если у вас возникнет желание воспользоваться в своих приложениях возможностями Kubernetes, это можно сделать с помощью оператора. Если оператор выйдет из строя, это не повлияет на доступность вашего приложения. Возникшую проблему нужно будет срочно устранить, однако ваши клиенты или конечные пользователи не должные ее заметить, что в корне меняет ситуацию.

Вариант с заменой узлов может быть очень полезным, если вы заранее создаете хорошо протестированные системные образы. Это позволит вам выделять новые серверы и сразу же присоединять их к кластеру. Такой процесс будет быстрым, так как все обновленное ПО, включая операционную систему и пакеты, уже установлено, а новые серверы могут развертываться с помощью тех же процедур, что и предыдущие.

Начинайте замену узлов кластера с плоскости управления. Если у вас есть отдельный кластер etcd, начать следует с него. Данные, которые ваш кластер хранит на постоянной основе, имеют большое значение и требуют осторожного обращения. Если проблема возникнет при обновлении первого же узла etcd, то процесс обновления можно будет относительно легко прервать, как следует подготовившись. Если же вы уже обновили все свои рабочие узлы и плоскость управления Kubernetes, и только потом столкнулись с проблемой обновления etcd, то откатывать весь процесс обратно будет непрактично — вам придется в срочном порядке исправлять проблему в активной системе. Вы потеряете возможность прервать все обновление целиком, собрать нужные ресурсы, заново все проверить и возобновить процесс позже. Вам нужно будет решить проблему незамедлительно или в крайнем случае убедиться в том, что имеющиеся версии можно оставить на какое-то время в их текущем виде.

Если у вас есть отдельный кластер etcd, попробуйте начинать замену с удаления старых узлов: сначала удалите имеющийся узел и только потом добавьте его обновленную замену, а не наоборот. Подобный метод позволяет оставить список узлов etcd без изменений. Если вы, к примеру, добавите четвертый узел в кластер etcd с тремя участниками, то придется обновить список всех узлов, что потребует перезагрузки. Удаление старого участника и добавление вместо него нового, по возможности с тем же IP-адресом, повлияет на работу кластера намного меньше. В документации etcd есть превосходное описание этого процесса, которое может побудить вас к обновлению etcd без прекращения работы. Для этого вам нужно будет по возможности применить тот же принцип к обновлению ОС и пакетов на серверах, но этот процесс обычно довольно приятный и совершенно безопасный.

Замену узлов плоскости управления можно начинать с добавления, применяя команду `kubeadm join` с флагом `--control-plane` к новым серверам, на которых установлены обновленные исполняемые файлы Kubernetes, включая kubeadm, kubectl и kubelet. По мере того, как новые узлы становятся доступными и подтверждают свою работоспособность, их старые версии можно отсоединять и затем удалять.

Если агенты etcd размещены на узлах плоскости управления, подтверждение работоспособности должно включать проверку etcd, а сами узлы должны содержать утилиту etcdctl для управления участниками кластера.

После этого можно переходить к замене рабочих узлов. Очередность добавления и удаления может быть любой, узлы можно заменять по одному или по нескольку за раз. Основной фактор здесь — загруженность кластера. Если ваш кластер находится под сильной нагрузкой, лучше сначала добавлять новые рабочие узлы, а затем отсоединять и удалять старые, чтобы у перемещенных Pod'ов было достаточно вычислительных ресурсов. Опять же, рекомендуется использовать системные образы, в которых уже установлено все обновленное программное обеспечение, и добавлять их в кластер с помощью команды `kubeadm join`. Как и прежде, это можно реализовать с помощью многих механизмов, которые уже применяются при развертывании кластера. На рис. 2.7 проиллюстрирована операция последовательной замены узлов плоскости управления и групповой замены рабочих узлов.



Рис. 2.7 Выполнение обновления путем замены узлов в кластере

Обновление без прекращения работы

Обновление без прекращения работы подходит для окружений с ограниченными ресурсами, в которых заменять узлы было бы непрактично. Это затрудняет процедуру отката, что в свою очередь повышает риски. Но данный недостаток можно и нужно смягчить за счет всеобъемлющего тестирования. Также имейте в виду, что Kubernetes в производственной конфигурации является системой с высокой доступностью. Если проводить такое обновление последовательно, по одному узлу за раз, риски снижаются. Поэтому, если для выполнения этапов этого процесса используется средство управления конфигурацией, такое как Ansible, то следует воздержаться от соблазна обновить сразу все промышленные узлы.

Если говорить об узлах etcd, то, согласно документации этого проекта, вы должны взять отдельный узел, отсоединить его, обновить ОС, etcd и другие пакеты, после

чего снова сделать его доступным. Если etcd выполняется в контейнере, то, прежде чем отсоединять участника кластера, можете заранее подготовить соответствующий образ, чтобы минимизировать время простоя.

Рабочие узлы и узлы плоскости управления Kubernetes должны обновляться с помощью kubeadm, если этот инструмент применялся для инициализации кластера. В документации есть подробные инструкции о том, как выполнять этот процесс для каждой минорной версии, начиная с 1.13. Рискуя прозвучать, как заезженная пластинка, повторим: готовьтесь к возможным сбоям, автоматизируйте все, что только можно, и проводите тщательное тестирование.

На этом мы заканчиваем обсуждение методов обновления. Теперь давайте вернемся к тому, с чего мы начали — какие механизмы потребуются, чтобы инициировать эти процессы выделения и обновления ресурсов. Мы оставили эту тему напоследок, поскольку она должна рассматриваться в контексте того, о чем мы уже поговорили в этой главе.

Механизмы инициирования

Мы обсудили все вопросы, связанные с моделью развертывания Kubernetes. Теперь будет полезно поговорить о механизмах, инициирующих автоматические процессы установки и администрирования. Они могут принимать разные формы. То, как вы инициируете сборку, масштабирование и обновление кластера, имеет большое значение, независимо от того, используете вы управляемый сервис Kubernetes, готовый установщик или собственную систему автоматизации, написанную с нуля.

У установщиков Kubernetes обычно имеются инструменты командной строки, с помощью которых можно инициировать процесс установки. Однако их использование в отрыве от всего остального лишает вас возможности задействовать единый источник информации и реестр компонентов кластера. Вам будет сложно управлять своим кластером, не имея списка его компонентов.

В последние годы стал популярным подход GitOps. В его рамках источником информации служит репозиторий исходного кода с конфигурацией для кластеров, которые вы администрируете. Когда в репозитории фиксируется конфигурация нового кластера, срабатывает автоматический механизм, который выделяет необходимые ресурсы. Когда обновляется существующая конфигурация, тот же механизм обновляет кластер, например, корректирует число рабочих узлов или устанавливает новые версии Kubernetes и дополнений.

Другой, более стандартный для Kubernetes подход состоит в представлении кластера и его зависимостей в виде пользовательских ресурсов и применении операторов Kubernetes, которые создают новые кластеры в соответствии с состоянием, объявленным в этих ресурсах. Этот метод реализован в таких проектах как Cluster API. Источником информации в этом случае выступают ресурсы Kubernetes, хранящиеся в etcd внутри управляющего кластера. Вместе с тем нередко выделяются отдельные управляющие кластеры для разных регионов и уровней. Подход GitOps можно

применять так, чтобы манифесты ресурсов кластера хранились в системе контроля версий и передавались подходящему управляющему кластеру с помощью конвейера. Этот вариант сочетает в себе лучшие качества GitOps и Kubernetes.

Резюме

Разрабатывая модель развертывания Kubernetes, как следует подумайте о том, какие управляемые сервисы или существующие установщики (бесплатные или коммерческие) вы можете для этого использовать. Автоматизация должна лежать в основе всех систем, которые вы создаете. Вы должны учитывать все архитектурные и топологические факторы, такие, как необычные требования, которые нужно соблюсти. Проанализируйте инфраструктурные зависимости и подумайте над тем, как их интегрировать в процесс развертывания. Взвесьте разные подходы к управлению серверами, из которых состоят ваши кластеры. Разберитесь в том, какие контейнерные компоненты будут составлять плоскость управления. Разработайте согласованные процедуры для установки дополнений, которые будут предоставлять неотъемлемые возможности вашей платформы приложений. Версионируйте свою платформу и позаботьтесь о последующих процессах администрирования и обновления, прежде чем развертывать в своих кластерах приложения.

ГЛАВА 3

Среда выполнения контейнеров

Платформа Kubernetes предназначена для оркестрации контейнеров. Однако сама по себе она не умеет их создавать, запускать и останавливать. Эти операции она делегирует подключаемому компоненту, известному как *среда выполнения контейнеров*. Эта среда отвечает за создание и администрирование контейнеров на узле кластера. В Linux она использует ряд механизмов ядра таких, как контрольные группы (cgroups) и пространства имен, для порождения процесса на основе образа контейнера. В сущности, Kubernetes (а если точнее, то kubelet) работает совместно с этой средой для управления контейнерами.

Как уже обсуждалось в *главе 1*, организации, разрабатывающие платформы поверх Kubernetes, должны принять ряд решений, одно из которых состоит в выборе среды выполнения контейнеров. Возможность выбора — это хорошо, она позволяет адаптировать платформу под ваши нужды, что открывает путь для инноваций и сложных сценариев, которые в противном случае не были бы возможными. С другой стороны, учитывая, насколько основополагающей является среда выполнения контейнеров, встает вопрос: почему ее реализация не входит в состав Kubernetes? Почему разработчики решили предоставить интерфейс для ее подключения и переложили эту ответственность на другой компонент?

Чтобы понять причину, мы сделаем краткий обзор истории контейнеров и объясним, как сложилось нынешнее положение вещей. Вначале речь пойдет об изобретении контейнеров и их влиянии на отрасль разработки программного обеспечения. В конце концов, без них проекта Kubernetes, скорее всего, не существовало бы. После этого мы обсудим инициативу Open Container Initiative (OCI), возникшую из-за необходимости в стандартизации сред выполнения контейнеров, образов и другого инструментария. Вы познакомитесь со спецификациями OCI и узнаете, какое отношение они имеют к Kubernetes. После OCI мы рассмотрим CRI (Container Runtime Interface — интерфейс среды выполнения контейнеров). Это связующее звено между kubelet и средой выполнения контейнеров, определяющее интерфейс, который такая среда должна реализовать, чтобы быть совместимой с Kubernetes. В конце мы расскажем, как выбрать среду выполнения контейнеров для своей платформы и проведем краткий обзор вариантов, доступных в экосистеме Kubernetes.

Появление контейнеров

Контрольные группы (cgroups) и пространства имен — это основные ингредиенты, необходимые для реализации контейнеров. Первые ограничивают количество ресурсов, доступных процессу (центральный процессор, память и т. д.), а вторые определяют, что этот процесс может "видеть" (т. е. подключенные разделы, процессы, се-

тевые интерфейсы и т. д.). Оба эти механизма существуют в ядре Linux с 2008 года. Пространства имен появились еще раньше. Так почему же контейнеры в их нынешнем виде стали популярны лишь спустя годы?

Чтобы ответить на этот вопрос, нужно сначала поговорить о состоянии дел в сфере ИТ и разработки ПО, которое сложилось на тот момент. Первое, на что следует обратить внимание, — это сложность приложений. В то время разработчики уже использовали сервис-ориентированные архитектуры и даже начали внедрять микросервисы. В результате организации получили различные преимущества, такие как удобство обслуживания, масштабируемость и возросшая производительность труда. Вместе с тем применение подобных архитектур привело к резкому увеличению числа компонентов, из которых состояли приложения. Проект, имеющий практическое применение, мог вполне включать в себя десяток сервисов, которые иногда были написаны на разных языках. Как можно себе представить, разработка и доставка таких приложений были (и продолжают быть) непростой задачей. Еще один фактор, о котором нужно помнить, состоит в том, что программное обеспечение быстро превратилось в конкурентное преимущество. Чем быстрее компания могла выпускать новые версии, тем привлекательней были ее продукты или услуги. Надежность развертывания ПО приобрела ключевое значение для организаций. Еще одним важным фактором стало появление общедоступных облаков, в которых можно было размещать свои приложения. Разработчики и команды эксплуатации должны были следить за тем, чтобы их программное обеспечение вело себя одинаково во всех окружениях: от ноутбука, на котором пишут код, до промышленного сервера, размещенного в чужом центре обработки данных.

Учитывая все это, нельзя не отметить, что индустрия созрела для инноваций. Так появился проект Docker, сделавший контейнеры общедоступными. Его авторы создали абстракцию, которая позволила разработчикам собирать и выполнять контейнеры с помощью удобной утилиты командной строки. Вместо того чтобы изучать низкоуровневые механизмы ядра, вы могли просто ввести в своем терминале команду `docker run`.

Контейнеры хоть и не оказались панацеей от всех бед, но улучшили многие стадии цикла разработки ПО. Во-первых, контейнеры и их образы позволили разработчикам описывать окружение приложения в коде. Им больше не нужно бороться с недостающими или неподходящими зависимостями. Во-вторых, контейнеры повлияли на процесс тестирования, сделав окружения воспроизводимыми. И последнее, контейнеры упростили развертывание программного обеспечения в промышленных условиях. Если в промышленной среде есть Docker Engine, то приложение можно развернуть с минимальными затруднениями. В целом контейнеры помогают организациям создавать ПО с нуля и доводить его до готовности к промышленному использованию более предсказуемым и эффективным образом.

Появление контейнеров также породило богатую экосистему со множеством разных инструментов, сред выполнения, реестров образов и многим другим. Эта экосистема была принята сообществом с энтузиазмом, но создала новую проблему: как обеспечить совместимость всех контейнерных решений между собой? В конце концов, инкапсуляция и гарантии переносимости являются одним из главных пре-

имуществ контейнеров. Чтобы ответить на этот вызов и улучшить внедрение контейнеров, участники рынка объединили усилия и совместно разработали под эгидой консорциума Linux Foundation открытую спецификацию: Open Container Initiative.

Open Container Initiative

С ростом популярности контейнеров стало очевидно, что для успеха этого движения нужны стандарты и спецификации. Open Container Initiative (OCI) — это открытый проект, основанный в 2015 году для совместной работы над спецификацией контейнерных технологий. Среди основателей этой инициативы можно выделить компанию Docker, которая передала в OCI утилиту runc, и проект CoreOS, который вывел среды выполнения контейнеров на следующий уровень, представив движок rkt.

OCI включает в себя три спецификации: для сред выполнения, для образов и для процесса дистрибуции. Они делают возможными разработку и инновации в области контейнеров и контейнерных платформ таких, как Kubernetes. Более того, одна из задач OCI состоит в том, чтобы конечные пользователи могли работать с переносимыми и совместимыми контейнерами, которые при необходимости можно было бы относительно легко применять в разных продуктах и решениях.

В следующих разделах мы исследуем спецификации сред выполнения и образов. Мы не станем подробно останавливаться на спецификации процесса дистрибуции, так как она в основном относится к реестрам образов контейнеров.

Спецификация OCI для сред выполнения

Спецификация OCI для сред выполнения определяет, как создавать экземпляры контейнеров и выполнять их совместимым с OCI образом. Прежде всего она описывает структуру конфигурации контейнера, включая такую информацию, как его корневая файловая система, команда для запуска, переменные окружения, учетная запись и группа, которые нужно использовать, лимиты на ресурсы и др. В листинге 3.1 приведен сокращенный пример конфигурационного файла, взятый из спецификации OCI для сред выполнения.

Листинг 3.1

```
{
    "ociVersion": "1.0.1",
    "process": {
        "terminal": true,
        "user": {
            "uid": 1,
            "gid": 1,
            "additionalGids": [
                5,
                6
```

```
            ]
        },
        "args": [
            "sh"
        ],
        "env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
            "TERM=xterm"
        ],
        "cwd": "/",
        ...
    },
    ...
    "mounts": [
        {
            "destination": "/proc",
            "type": "proc",
            "source": "proc"
        },
    ...
    },
    ...
}
```

Спецификация сред выполнения также определяет операции, которые должны поддерживать эти среды, включая команды `create` (создать), `start` (запустить), `kill` (остановить), `delete` (удалить) и `state` (состояние); последняя предоставляет информацию о состоянии контейнера. Помимо этого спецификация описывает жизненный цикл контейнера и его прохождение через разные стадии: (1) `creating` — период, когда среда выполнения создает контейнер; (2) `created` — когда среда выполнения завершила операцию `create`; (3) `running` — когда процесс контейнера запустился и выполняется; (4) `stopped` — когда процесс контейнера завершил работу.

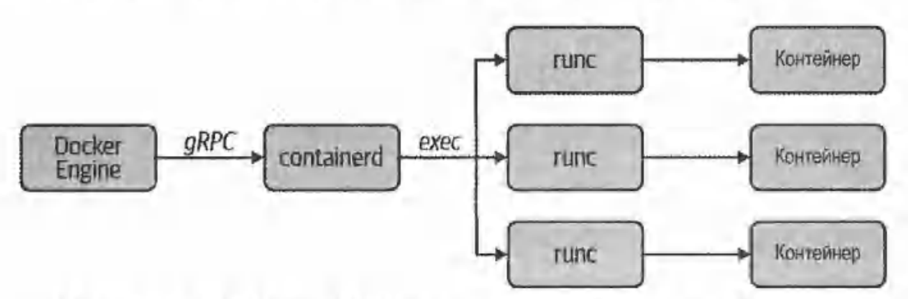


Рис. 3.1. Docker Engine, containerd и другие среды выполнения используют runc для создания экземпляров контейнеров в соответствии со спецификацией OCI

В состав проекта OCI также входит *runc*, низкоуровневая среда выполнения контейнеров, которая реализует спецификацию OCI. Как видно на рис. 3.1, среды вы-

полнения более высокого уровня, такие как Docker и CRI-O, используют runc для создания экземпляров контейнеров в соответствии со спецификацией. Применение runc позволяет разработчикам сред выполнения контейнеров сосредоточиться на более высокоуровневых возможностях, таких как скачивание образов, настройка сети, работа с хранилищем данных — и все это в соответствии со спецификацией OCI.

Спецификация OCI для образов

Эта спецификация предназначена для образов контейнеров. Она определяет манифест, необязательный индекс образа, набор слоев файловой системы и конфигурацию. Манифест описывает образ; он содержит ссылку на конфигурацию образа, список его слоев и необязательный словарь аннотаций. В листинге 3.2 приведен пример манифеста, взятый из спецификации OCI для образов.

Листинг 3.2

```
{
  "schemaVersion": 2,
  "config": {
    "mediaType": "application/vnd.oci.image.config.v1+json",
    "size": 7023,
    "digest": "sha256:b5b2b2c507a0944348e0303114d8d93aaaa081732b86451d9bce1f4..."
  },
  "layers": [
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "size": 32654,
      "digest": "sha256:9834876dcfb05cb167a5c24953eba58c4ac89b1adf57f28f2f9d0..."
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "size": 16724,
      "digest": "sha256:3c3a4604a545cdc127456d94e421cd355bca5b528f4a9c1905b15..."
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "size": 73109,
      "digest": "sha256:ec4b8955958665577945c89419d1af06b5f7636b4ac3da7f12184..."
    }
  ],
  "annotations": {
    "com.example.key1": "value1",
    "com.example.key2": "value2"
  }
}
```

Индекс образа — это манифест высшего уровня, который позволяет создавать образы контейнеров с поддержкой разных платформ. Он содержит ссылки на манифест каждой платформы. В листинге 3.3 приведен пример, взятый из спецификации. Обратите внимание на то, как этот индекс ссылается на два разных манифеста: один для `ppc64le/linux`, а другой — для `amd64/linux`.

Листинг 3.3

```
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "size": 7143,
      "digest": "sha256:e692418e4cbaf90ca69d05a66403747baa33ee08806650b51fab...",
      "platform": {
        "architecture": "ppc64le",
        "os": "linux"
      }
    },
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "size": 7682,
      "digest": "sha256:5b0bcabd1ed22e9fb1310cf6c2dec7cdef19f0ad69efa1f392e9...",
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      }
    }
  ],
  "annotations": {
    "com.example.key1": "value1",
    "com.example.key2": "value2"
  }
}
```

Каждый манифест образа OCI ссылается на соответствующую конфигурацию, которая состоит из точки входа в образ, команды, рабочей директории, переменных окружения и пр. Среда выполнения использует эту конфигурацию при создании экземпляра контейнера из образа. В листинге 3.4 приведен пример конфигурации для образа контейнера. Некоторые поля были опущены для краткости.

Листинг 3.4

```
{
  "architecture": "amd64",
```

```
"config": {
  ...
  "ExposedPorts": {
    "53/tcp": {},
    "53/udp": {}
  },
  "Tty": false,
  "OpenStdin": false,
  "StdinOnce": false,
  "Env": [
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
  ],
  "Cmd": null,
  "Image": "sha256:7ccecf40b555e5ef2d8d3514257b69c2f4018c767e7a20dbaf4733...",
  "Volumes": null,
  "WorkingDir": "",
  "Entrypoint": [
    "/coredns"
  ],
  "OnBuild": null,
  "Labels": null
},
"created": "2020-01-28T19:16:47.907002703Z",
...
```

Спецификация OCI для образов также описывает процесс создания и администрирования слоев образа контейнера. Слои фактически представляют собой TAR-архивы с файлами и директориями. Спецификация определяет разные типы слоев, включая несжатые, сжатые с помощью утилиты `gzip` и не предназначенные для распространения. Каждый слой имеет уникальный идентификатор, в качестве которого обычно используется контрольная сумма его содержимого в формате SHA256. Как уже обсуждалось ранее, манифест образа контейнера ссылается на один или несколько слоев. В качестве ссылок на тот или иной слой применяется хеш SHA256. Файловая система итогового образа контейнера является результатом применения каждого из слоев, перечисленных в манифесте.

Спецификация OCI для образов имеет ключевое значение, позволяя сделать образы контейнеров совместимыми с разными инструментами и контейнерными платформами. С ее помощью разработчики создают различные инструменты: kaniko и Buildah — для создания контейнеров в пространстве пользователя, Jib — для контейнеров на основе Java и Cloud Native Buildpacks — для упрощения и автоматизации сборки (некоторые из них мы исследуем в *главе 15*). В целом данная спецификация гарантирует, что Kubernetes может выполнять контейнеры, независимо от того, с помощью какого инструментария они созданы.

Интерфейс среды выполнения контейнеров

Как уже обсуждалось в предыдущих главах, Kubernetes предлагает множество возможностей для расширения, которые позволят вам создать собственную платформу приложений. Один из ключевых примеров — интерфейс среды выполнения контейнеров (англ. Container Runtime Interface или CRI). Он появился в Kubernetes версии 1.5 для поддержки развивающейся экосистемы сред выполнения контейнеров, в том числе rkt от CoreOS и среды на основе гипервизора Clear Containers от Intel (позже была переименована в Kata Containers).

Прежде чем появился CRI, добавление поддержки новой среды выполнения контейнеров требовало выпуска новой версии Kubernetes и глубоких знаний кодовой базы этого проекта. Теперь же для обеспечения совместимости среды выполнения с Kubernetes разработчикам достаточно просто соблюсти этот интерфейс.

Интерфейс CRI в целом создавался для того, чтобы скрыть детали реализации среды выполнения контейнеров от Kubernetes (если точнее, от kubelet). Это классический пример принципа инверсии зависимостей. Реализация kubelet, которая раньше содержала код, относящийся к определенным средам выполнения контейнеров, и множество условных выражений, стала более лаконичной и начала полагаться на интерфейс CRI. Таким образом, реализацию kubelet удалось упростить и при этом сделать ее более расширяемой и поддающейся тестированию.

Интерфейс CRI реализован с использованием gRPC и Protocol Buffers. Он определяет два сервиса, *RuntimeService* и *ImageService*, с помощью которых kubelet взаимодействует со средой выполнения контейнеров. RuntimeService отвечает за все операции, связанные с Pod'ами, включая создание, запуск и остановку контейнеров, удаление Pod'ов и т. д. ImageService берет на себя операции с образами контейнеров, позволяя выводить их списки, скачивать и удалять из узлов.

Мы могли бы подробно рассмотреть в этой главе интерфейсы RuntimeService и ImageService, но вам будет полезней разобраться в том, как работает, наверное, самая полезная операция в Kubernetes: запуск Pod'а на узле. Поэтому давайте поговорим в следующем разделе о том, как kubelet взаимодействует со средой выполнения контейнеров через интерфейс CRI.

Запуск Pod'а



Приведенный далее материал основан на Kubernetes v1.18.2 и containerd v1.3.4. Эти компоненты используют CRI версии v1alpha2.

После того как выполнение Pod'а запланировано на узле, ее запуск производится совместными усилиями kubelet и среды выполнения контейнеров. Как уже упоминалось, kubelet взаимодействует с этой средой через CRI. В данном случае мы исследуем взаимодействие kubelet с подключаемым модулем CRI для containerd.

Подключаемый модуль CRI для containerd запускает сервер gRPC, который прослушивает Unix-сокет. По умолчанию этот сокет имеет путь /run/containerd/containerd.sock.

Агент kubelet сконфигурирован для взаимодействия с containerd через этот сокет с помощью флагов командной строки `container-runtime` и `container-runtime-endpoint`:

```
/usr/bin/kubelet
  --container-runtime=remote
  --container-runtime-endpoint=/run/containerd/containerd.sock
  ... другие флаги ...
```

Чтобы запустить Pod, kubelet сначала создает для него изолированное окружение с помощью метода `RunPodSandbox` из RuntimeService. Поскольку Pod может состоять из одного или более контейнеров, нам сначала необходимо создать изолированное окружение, чтобы подготовить для каждого контейнера общее сетевое пространство имен Linux (помимо прочего). При вызове этого метода kubelet передает среде containerd метаданные и конфигурацию, включая название Pod'а, уникальный идентификатор, пространство имен Kubernetes, конфигурацию DNS и т.д. Чтобы агент kubelet мог создавать контейнеры в только что созданном изолированном окружении, среда выполнения передает ему идентификатор этого окружения.

Когда изолированное окружение станет доступным, kubelet проверяет, присутствует ли образ контейнера на узле, используя метод `ImageStatus` из ImageService. Метод `ImageStatus` возвращает информацию об образе. Если образ отсутствует, возвращается null, и kubelet пытается его скачать. Для этого вызывается метод `PullImage` из ImageService. Скачав образ, среда выполнения возвращает его хеш в формате SHA256, который kubelet использует для создания контейнера.

После создания изолированного окружения и скачивания образа агент kubelet создает контейнеры с помощью метода `CreateContainer` из RuntimeService. Для этого среде выполнения предоставляется идентификатор окружения и конфигурация контейнера. Последняя включает в себя всю информацию, которую можно было бы ожидать, в том числе хеш образа контейнера, команду и аргументы, переменные окружения, подключенные разделы и т. д. В процессе создания контейнера среда выполнения генерирует его идентификатор, который затем возвращается агенту kubelet; именно он отображается в поле с состоянием Pod'а в разделе `containerStatuses`:

```
containerStatuses:
  - containerID: containerd://0018556b01e1662c5e7e2dcddb2bb09d0edff6cf6933...
    image: docker.io/library/nginx:latest
```

Дальше агент kubelet переходит к запуску контейнера с помощью метода `StartContainer` из RuntimeService. При вызове этого метода используется идентификатор, полученный от среды выполнения.

Вот и все! В этом разделе вы узнали, как kubelet взаимодействует со средой выполнения контейнеров посредством CRI. В частности, мы рассмотрели методы gRPC, вызываемые при запуске Pod'а, включая те, которые принадлежат сервисам ImageService и RuntimeService. Оба эти CRI-сервиса предоставляют дополнительные методы, с помощью которых kubelet выполняет другие задачи. Помимо операций для управления Pod'ами и контейнерами (таких как создание, чтение, модификация и удаление), CRI также определяет методы для выполнения команд внутри

контейнеров (`Exec` и `ExecSync`), присоединения к контейнеру (`Attach`), перенаправления определенных портов контейнера (`PortForward`) и др.

Выбор среды выполнения

Учитывая доступность CRI, разработчики платформ имеют широкий выбор сред выполнения контейнеров, хотя в реальности за последние несколько лет эти среды превратились в один из аспектов реализации. Если вы используете дистрибутив или управляемый сервис Kubernetes, значит, данную среду, скорее всего, для вас выбрал кто-то другой. Это относится даже к проектам сообщества, например, Cluster API предоставляет заранее подготовленные образы узлов с уже включенной средой выполнения контейнеров.

Тем не менее, если у вас есть возможность выбора, или вам нужна специализированная среда выполнения (например, основанная на ВМ), вы должны иметь в своем распоряжении всю информацию, необходимую для принятия этого решения. В данном разделе мы обсудим факторы, на которые следует обратить внимание при выборе среды выполнения контейнеров.

Первый вопрос, который мы любим задавать, когда помогаем организациям, звучит так: с какой средой выполнения контейнеров вы уже имели дело? В большинстве случаев организации, имеющие богатый опыт работы с контейнерами, используют Docker и уже знакомы с соответствующим инструментарием. Несмотря на поддержку Docker в Kubernetes, мы не рекомендуем эту среду, так как она обладает широким набором возможностей, которые в Kubernetes не нужны, например, создание образов, объединение контейнеров в сеть и т. д. Иными словами, полноценная программа Docker слишком тяжеловесная или раздутая для тех задач, в которых она применяется в Kubernetes. Но хорошая новость состоит в том, что внутри Docker использует containerd, — одну из самых распространенных сред выполнения контейнеров. С другой стороны, администраторам платформы необходимо научиться работать с ее интерфейсом командной строки.

Еще один аспект, о котором нужно подумать, — наличие поддержки среды выполнения контейнеров. Это зависит от того, в каком виде вы используете Kubernetes. Дистрибутивы Kubernetes, такие как Tanzu Kubernetes Grid от VMware, OpenShift от RedHat и др., обычно поставляются с определенной средой. Менять ее следует только по крайне уважительной причине. В этом случае убедитесь, что вы понимаете, как выбор другой среды выполнения контейнеров скажется на поддержке.

Еще один фактор, имеющий непосредственное отношение к поддержке, — тестирование среды выполнения контейнеров на соответствие. Проект Kubernetes, а точнее специальная группа sig-node (Node Special Interest Group), определяет набор проверок CRI и тестов соответствия узлов, которые позволяют убедиться в том, что среда совместима и ведет себя предсказуемо. Эти тесты входят в каждый выпуск Kubernetes, и код разных сред может иметь различную степень покрытия. Как несложно догадаться, чем обширней покрытие тестами, тем лучше, так как это позволяет выявить любые проблемы со средой выполнения контейнеров в процессе под-

готовки выпуска Kubernetes. Сообщество публикует все тесты и их результаты на веб-сайте Kubernetes Test Grid (**https://k8s-testgrid.appspot.com**). При выборе среды выполнения следует учитывать, насколько хорошо она проходит тестирование на соответствие, и, в более широком контексте, какое отношение она имеет с проектом Kubernetes в целом.

Наконец, вы должны определить, нужны ли вашим рабочим заданиям более строгие гарантии изоляции по сравнению с теми, которые предоставляют контейнеры Linux. Не так часто, но все же встречаются сценарии использования, в которых рабочие задания должны иметь изоляцию уровня ВМ, как в случае с выполнением непроверенного кода или приложений, требующих строгих гарантий мультиарендности. В таких ситуациях можно воспользоваться специализированными средами выполнения наподобие Kata Containers.

Итак, мы обсудили факторы, которые нужно учитывать при выборе среды выполнения контейнеров. Теперь давайте рассмотрим наиболее распространенные среды: Docker, containerd и CRI-O. Мы также исследуем Kata Containers, чтобы вы понимали, как выполнять Pod-оболочки в виртуальных машинах вместо контейнеров Linux. В заключение речь пойдет о Virtual Kubelet, который предоставляет альтернативный способ выполнения рабочих заданий в Kubernetes, но при этом не является ни средой выполнения контейнеров, ни реализацией CRI.

Docker

Kubernetes поддерживает Docker Engine в качестве среды выполнения контейнеров за счет наличия промежуточного слоя CRI под названием *dockershim*. Этот слой представляет собой компонент, встроенный в kubelet. Это фактически gRPC-сервер, который реализует сервисы CRI, описанные ранее в этой главе. Вместо того чтобы отдельно реализовывать в kubelet поддержку CRI и Docker Engine, dockershim играет роль внешнего интерфейса, который позволяет взаимодействовать с Docker через CRI. dockershim берет на себя преобразование между вызовами CRI и API-интерфейса Docker Engine. На рис. 3.2 показано, как kubelet использует этот промежуточный слой для работы с Docker.

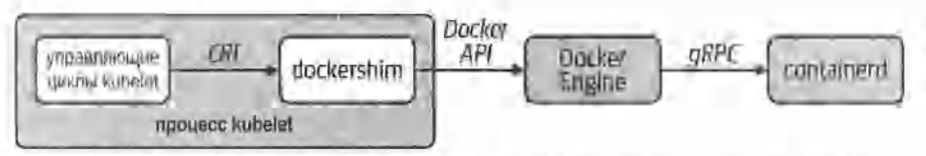


Рис. 3.2. Взаимодействие между kubelet и Docker Engine посредством dockershim

Как уже упоминалось ранее в этой главе, внутри Docker используется containerd. В связи с этим containerd в конечном счете принимает все входящие API-вызовы, которые приходят от kubelet. Поэтому контейнер в итоге создается в иерархии процессов containerd, а не Docker:

```
systemd
  └─containerd
```

```
    └─containerd-shim -namespace moby -workdir ...
        └─nginx
            └─nginx
```

Если говорить о диагностике, то Docker CLI позволяет выводить список контейнеров, запущенных на заданном узле, и анализировать каждый из них. У Docker нет такого понятия как Pod, однако dockershim формирует название контейнера из пространства имен Kubernetes, а также имени и идентификатора Pod'а. Например, в следующем фрагменте кода показаны контейнеры, принадлежащие Pod'у `nginx` в пространстве имен `default`. Контейнер инфраструктуры Pod'а (он же pause container) имеет название с префиксом `k8s_POD_`:

```
$ docker ps --format='{{.ID}}\t{{.Names}}' | grep nginx_default
3c8c01f47424    k8s_nginx_nginx_default_6470b3d3-87a3-499c-8562-d59ba27bced5_3
c34ad8d80c4d    k8s_POD_nginx_default_6470b3d3-87a3-499c-8562-d59ba27bced5_3
```

Анализ контейнеров можно выполнить посредством утилиты `ctr` командной строки containerd, хотя ее вывод не настолько наглядный, как у Docker CLI. Docker Engine использует пространство имен containerd под названием `moby`:

```
$ ctr --namespace moby containers list
CONTAINER                   IMAGE    RUNTIME
07ba23a409f31bec7f163a...   -        io.containerd.runtime.v1.linux
0bfc5a735c213b9b296dad...   -        io.containerd.runtime.v1.linux
2d1c9cb39c674f75caf595...   -        io.containerd.runtime.v1.linux
...
```

Наконец, вы можете воспользоваться утилитой `crictl`, если она присутствует на узле. Это инструмент командной строки, разработанный сообществом Kubernetes, который представляет собой консольный клиент для взаимодействия со средами выполнения контейнеров через интерфейс CRI. Несмотря на то, что в Docker нет реализации CRI, `crictl` можно сочетать с Unix-сокетом `dockershim`:

```
$ crictl --runtime-endpoint unix:///var/run/dockershim.sock ps --name nginx
CONTAINER       ID IMAGE              CREATED         STATE     NAME    POD ID
07ba23a409f31   nginx@sha256:b0a...   3 seconds ago   Running   nginx   ea179944...
```

containerd

containerd — это, наверное, самая распространенная среда выполнения контейнеров, которая нам встречалась при создании платформ на основе Kubernetes. На момент написания этих строк среда containerd используется по умолчанию в образах узлов, основанных на Cluster API, и доступна в различных управляемых версиях Kubernetes (например, AKS, EKS и GKE).

Среда выполнения контейнеров containerd реализует CRI с помощью подключаемого модуля. Это стандартный модуль, который входит в состав containerd, начиная с версии 1.1, и включен по умолчанию. API-интерфейсы containerd, основанные на протоколе gRPC, доступны посредством Unix-сокета через путь /run/containerd/containerd.sock. Если говорить об управлении Pod-оболочками, то kubelet использует этот сокет для взаимодействия с containerd, как показано на рис. 3.3.

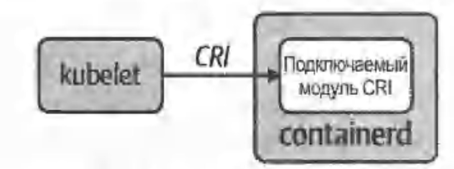


Рис. 3.3. Взаимодействие между containerd и kubelet через подключаемый модуль CRI, принадлежащий containerd

Иерархия процессов созданных контейнеров выглядит точно так же, как и в случае Docker Engine. Этого следовало ожидать, ведь Docker Engine применяет containerd для управления контейнерами:

```
systemd
  └─containerd
      └─containerd-shim -namespace k8s.io -workdir ...
          └─nginx
              └─nginx
```

Чтобы проанализировать контейнер на узле, можно воспользоваться утилитой `ctr`, интерфейсом командной строки containerd. Контейнеры, управляемые с помощью Kubernetes, находятся в пространстве имен containerd под названием `k8s.io`, а не `moby`, как в случае с Docker:

```
$ ctr --namespace k8s.io containers ls | grep nginx
c85e47fa...    docker.io/library/nginx:latest    io.containerd.runtime.v1.linux
```

Вы также можете использовать консольную утилиту `crictl` для взаимодействия с containerd через Unix-сокет:

```
$ crictl --runtime-endpoint unix:///run/containerd/containerd.sock ps
  --name nginx
CONTAINER ID    IMAGE          CREATED          STATE      NAME    POD ID
c85e47faf3616   4bb46517cac39  39 seconds ago   Running    nginx   73caea404b92a
```

CRI-O

CRI-O — это среда выполнения контейнеров, разработанная специально для Kubernetes. Как, наверное, можно догадаться по названию, она представляет собой реализацию CRI. В связи с этим, в отличие от Docker и containerd, она не предназначена для использования вне Kubernetes. На момент написания этих строк одним из основных потребителей CRI-O является платформа RedHat OpenShift.

Подобно containerd, CRI-O предоставляет доступ к CRI через Unix-сокет. Kubelet использует этот сокет, который обычно имеет путь /var/run/crio/crio.sock, для взаимодействия с CRI-O (рис. 3.4).

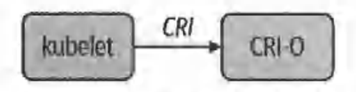


Рис. 3.4. Взаимодействие между kubelet и CRI-O с помощью API-интерфейсов CRI

При создании экземпляров контейнеров CRI-O порождает процесс под названием *common*. Это средство мониторинга контейнеров, которое является их родительским процессом и предоставляет доступ к таким операциям, как присоединение к контейнеру, сохранение его потоков STDOUT и STDERR в журнальные файлы и завершение его работы:

```
systemd
  └─conmon -s -c ed779... -n k8s_nginx_nginx_default_e9115... -u8cdf0c...
      └─nginx
          └─nginx
```

Поскольку среда CRI-O была создана в качестве низкоуровневого компонента Kubernetes, у нее нет интерфейса командной строки. Тем не менее, для работы с ней, как и с любой другой средой выполнения контейнеров, реализующей CRI, можно использовать утилиту `crictl`:

```
$ crictl --runtime-endpoint unix:///var/run/crio/crio.sock ps --name nginx
CONTAINER   IMAGE                 CREATED         STATE     NAME    POD ID
8cdf0c...   nginx@sha256:179...   2 minutes ago   Running   nginx   eabf15237...
```

Kata Containers

Kata Containers — это открытая специализированная среда выполнения, которая выполняет приложения не в контейнерах, а в легковесных виртуальных машинах. Этот проект имеет богатую историю и является результатом слияния двух других сред выполнения на основе ВМ: Clear Containers от Intel и RunV от Hyper.sh.

Из-за применения виртуальных машин Kata предлагает более строгие гарантии изоляции, чем контейнеры Linux. Если ваши требования к безопасности исключают возможность выполнения рабочих заданий на одном и том же ядре Linux, или же изоляция cgroup не позволяет обеспечить нужный уровень гарантий, Kata Containers может быть хорошим решением. Например, контейнеры Kata часто применяют для создания мультитенантных кластеров Kubernetes, в которых выполняется непроверенный программный код. Облачные провайдеры, такие как Baidu Cloud (**https://oreil.ly/btDL9**) и Huawei Cloud (**https:// oreil.ly/Mzarh**), используют Kata Containers в инфраструктуре своих облаков.



Рис. 3.5. Взаимодействие между kubelet и Kata Containers посредством containerd

Для совместной работы Kata Containers и Kubernetes необходима подключаемая среда выполнения контейнеров, которая будет играть роль промежуточного слоя между kubelet и средой Kata (рис. 3.5). Дело в том, что проект Kata Containers не реализует CRI. Вместо этого взаимодействие с Kubernetes осуществляется через

существующие среды выполнения контейнеров такие, как containerd. Для интеграции со средой containerd в Kata Containers реализован ее API-интерфейс, а именно containerd-shim API v2 (**https://oreil.ly/DxGyZ**).

Поскольку среда containerd, которая нужна для Kata Containers, доступна на узлах, мы можем разместить на одном и том же узле Pod'ы, основанные как на контейнерах, так и на ВМ. Kubernetes предоставляет механизм под названием Runtime Class, предназначенный для конфигурации и запуска разных сред выполнения контейнеров. Используя API-интерфейс этого механизма, вы можете предлагать разные среды выполнения в рамках единой платформы Kubernetes, позволяя разработчикам выбрать ту среду, которая лучше отвечает их потребностям. В следующем фрагменте кода показан пример Runtime Class для среды Kata Containers:

```
apiVersion: node.k8s.io/v1beta1
kind: RuntimeClass
metadata:
    name: kata-containers
handler: kata
```

Чтобы использовать среду выполнения Kata Containers, разработчики должны указать имя соответствующего класса в спецификации Pod'a:

```
apiVersion: v1
kind: Pod
metadata:
  name: kata-example
spec:
  containers:
  - image: nginx
    name: nginx
  runtimeClassName: kata-containers
```

Проект Kata Containers поддерживает выполнение приложений с помощью разных гипервизоров, включая QEMU (**https://www.qemu.org**), NEMU (**https://github.com/intel/nemu**) и AWS Firecracker (**https://firecracker-microvm.github.io**). Например, если использовать QEMU, то после запуска Pod'a на основе среды выполнения класса `kata-containers` можно видеть соответствующий процесс:

```
$ ps -ef | grep qemu
root     38290     1  0 16:02 ?        00:00:17
   /snap/kata-containers/690/usr/bin/qemu-system-x86_64
   -name sandbox-c136a9addde4f26457901ccef9de49f02556cc8c5135b091f6d36cfc97...
   -uuid aaae32b3-9916-4d13-b385-dd8390d0daf4
   -machine pc,accel=kvm,kernel_irqchip
   -cpu host
   -m 2048M,slots=10,maxmem=65005M
   ...
```

Проект Kata Containers предлагает интересные возможности, но мы считаем его нишевым, так как в реальных условиях он нам не встречался. Тем не менее, если вам в вашем кластере Kubernetes нужны гарантии изоляции уровня ВМ, Kata Containers заслуживает внимания.

Virtual Kubelet

Virtual Kubelet (**https://github.com/virtual-kubelet/virtual-kubelet**) — это агент с открытым исходным кодом, который ведет себя так же, как kubelet, но при этом имеет внутреннюю часть с расширяемым API-интерфейсом. Сам по себе он не является средой выполнения контейнеров, однако его главное назначение — сделать возможным использование альтернативных сред для выполнения Pod-оболочек Kubernetes. Благодаря расширяемой архитектуре Virtual Kubelet этими альтернативными средами могут быть фактически любые системы, способные выполнять приложения, включая платформы бессерверных и граничных вычислений. Например, как показано на рис. 3.6, Virtual Kubelet может запускать Pod'ы в таких облачных сервисах, как Azure Container Instances и AWS Fargate.

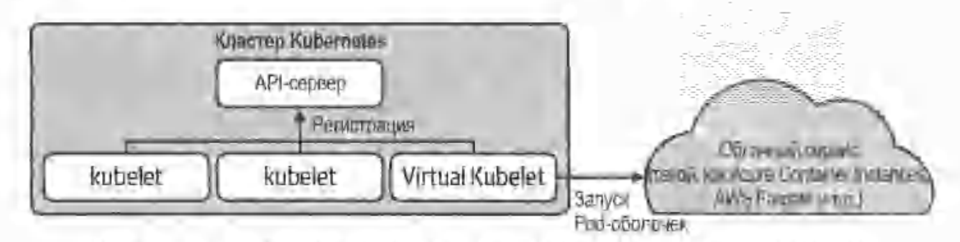


Рис. 3.6. Virtual Kubelet с Pod-оболочками, запущенными в облачном сервисе, таком как Azure Container Instances, AWS Fargate и т. д.

Сообщество Virtual Kubelet предлагает поддержку разнообразных провайдеров, которые можно выбирать в зависимости от ваших нужд, включая AWS Fargate, Azure Container Instances, HashiCorp Nomad и др. Если у вас есть более специфический сценарий использования, можно реализовать собственный провайдер. Для этого нужно написать программу на языке Go с помощью библиотек Virtual Kubelet для организации взаимодействия с Kubernetes и выполнения таких операций, как регистрация узлов, запуск Pod'ов, экспорт API-интерфейсов, которые ожидает Kubernetes, и т. д.

Несмотря на интересные возможности, которые предлагает Virtual Kubelet, мы еще не встречали реальной ситуации, в которой нельзя было бы обойтись без этого проекта. Тем не менее, о нем следует знать, и вы должны иметь его в своем арсенале.

Резюме

Среда выполнения контейнеров — фундаментальный компонент платформы, основанной на Kubernetes. В конце концов, без такой среды невозможно выполнять кон-

тейнерные приложения. Как вы узнали в этой главе, Kubernetes использует CRI (Container Runtime Interface — интерфейс среды выполнения контейнеров) для взаимодействия со средой выполнения контейнеров. Одно из главных преимуществ CRI состоит в его расширяемости, благодаря которой вы можете отдать предпочтение тем средам, которые лучше всего отвечают вашим нуждам. Чтобы вы имели представление о разных средах выполнения контейнеров, доступных в экосистеме Kubernetes, мы обсудили те из них, которые обычно встречаются нам в реальных условиях, такие как Docker, containerd и пр. Знакомство с разными инструментами и дальнейшее исследование их возможностей должно помочь вам в выборе среды выполнения контейнеров, которая удовлетворяет требованиям вашей платформы приложений.

ГЛАВА 4

Хранилище данных контейнера

Платформа Kubernetes изначально была рассчитана на приложения с неизменным состоянием, однако выполнение сервисов, которые хранят свое состояние, становится все более распространенным. Даже такое сложное программное обеспечение, как базы данных и очереди сообщений, начинает использоваться в кластерах Kubernetes. Для его поддержки требуются механизмы постоянного хранения данных. В частности, речь идет о системах, способных обеспечить повышенную устойчивость и доступность в различных неблагоприятных условиях, таких как сбой приложения или перенос на другой хост.

В этой главе мы поговорим о том, каким образом наша платформа может предоставлять приложениям услуги по хранению данных. Вначале будут рассмотрены ключевые аспекты постоянного хранения и характеристики, которые мы ожидаем от хранилища. Затем мы обсудим соответствующие компоненты, доступные в Kubernetes. Когда речь пойдет о повышенных требованиях к системе хранения, мы обратим наше внимание на интерфейс хранилищ для контейнеров (Container Storage Interface или CSI; **https://kubernetes-csi.github.io/docs**), который делает возможным интеграцию с разными провайдерами. В заключение мы исследуем применение подключаемых модулей CSI для предоставления нашим приложениям хранилища с возможностью самообслуживания.



Хранение данных само по себе является обширной темой. Мы хотим дать вам ровно столько подробностей, сколько требуется для обоснованного выбора хранилища, которое вы можете предоставлять своим приложениям. Если у вас нет опыта в данной сфере, мы настоятельно рекомендуем вам обсудить эти вопросы с вашей командой, отвечающей за инфраструктуру или хранение данных. Kubernetes не избавляет вашу организацию от необходимости в специалистах, которые ориентируются в этой области.

Требования к хранилищу

Прежде чем переходить к способам и вариантам организации хранения данных в Kubernetes, следует вернуться немного назад и проанализировать ключевые характеристики, которыми должно обладать хранилище. На уровне инфраструктуры и приложения следует обращать внимание на такие моменты:

- режимы доступа;
- расширение объема томов;
- динамическое выделение;
- резервное копирование и восстановление;

- блочное, файловое и объектное хранение;
- временные данные;
- выбор провайдера.

Режимы доступа

Существуют три режима доступа, поддержку которых можно предоставлять приложениям:

- *ReadWriteOnce (RWO)* — чтение и запись тома одним Pod'ом.
- *ReadOnlyMany (ROX)* — чтение тома несколькими Pod'ами.
- *ReadWriteMany (RWX)* — чтение и запись тома несколькими Pod'ами.

В подавляющем большинстве облачно-ориентированных приложений применяется RWO. Это единственный режим, предлагаемый такими популярными провайдерами, как Amazon Elastic Block Storage (EBS) (**https://aws.amazon.com/ebs**) и Azure Disk Storage (**https://oreil.ly/wAtBg**), так как диск можно присоединить только к одному узлу. Это ограничение может показаться проблематичным, однако большинство облачно-ориентированных приложений лучше всего работают именно с такими хранилищами, в которых том обеспечивает высокопроизводительные операции чтения/записи и эксклюзивный доступ.

Нам много раз встречались устаревшие приложения, которым нужна поддержка RWX. Зачастую они написаны с расчетом на доступ к сетевой файловой системе (англ. Network File System или NFS; **https://oreil.ly/OrsBR**). Но и в случае, когда сервисам нужно разделять общее состояние, существуют более изящные решения, такие, как очереди сообщений или базы данных. Кроме того, если приложению нужно поделиться данными, это обычно лучше всего делать посредством API-интерфейса, а не за счет предоставления доступа к собственной файловой системе. В связи с этим многие сценарии использования RWX являются сомнительными. Перед авторами платформы может стоять непростой выбор: предоставить хранилище, совместимое с RWX, или попросить разработчиков заново спроектировать свои приложения (если только NFS не является правильным выбором с точки зрения архитектуры). Если вы решите сделать поддержку ROX или RWX обязательной, у вас будет возможность интегрироваться с несколькими провайдерами, такими, как Amazon Elastic File System (EFS) (**https://aws.amazon.com/efs**) и Azure File Share (**https://oreil.ly/u6lIiQ**).

Расширение томов

Со временем приложение может заполнить свой том. Это проблема, так как замена тома на более вместительный требует переноса данных. Решением может быть поддержка расширения томов. В контексте средства оркестрации контейнеров такого как Kubernetes, процесс расширения состоит из нескольких этапов:

1. Запросить дополнительное место у оркестратора (например, с помощью средства PersistentVolumeClaim).

2. Увеличить размер тома с использованием провайдера хранилища.
3. Расширить файловую систему, чтобы воспользоваться более емким томом.

По завершении этого процесса Pod получит доступ к дополнительному месту. Реализация описанного подхода зависит от того, какая система хранения используется внутри, и позволяет ли интеграция в Kubernetes выполнить приведенные шаги. Пример расширения тома будет рассмотрен позже в этой главе.

Выделение томов

Вам доступны две модели выделения томов: динамическая и статическая. Статическое выделение подразумевает, что тома, с которыми работает Kubernetes, создаются на узлах. Динамическое выделение — это когда в кластере есть драйвер, который принимает запросы на выделение хранилища от приложений и удовлетворяет их, взаимодействуя с соответствующим провайдером. Из этих двух моделей по возможности следует отдавать предпочтение динамической. Зачастую выбор модели продиктован наличием у вашей внутренней системы хранения данных драйвера, совместимого с Kubernetes. Мы подробно обсудим эти драйверы чуть позже.

Резервное копирование и восстановление

Резервное копирование — это один из сложнейших аспектов хранения данных, особенно если требуется автоматическое восстановление. Говоря в общих чертах, резервная копия создается на случай потери данных. Обычно стратегии резервного копирования балансируются гарантиями доступности наших систем хранения. Например, несмотря на свою значимость, резервные копии могут быть не настолько важными, если система хранения предоставляет гарантии репликации, согласно которым неисправность оборудования *не* приводит к потере данных. Еще один фактор состоит в том, что приложениям могут быть нужны разные процедуры для резервного копирования и восстановления. Мысль о том, что мы можем создать резервную копию всего кластера и восстановить из нее нашу систему, как правило, является наивной, так как для ее воплощения как минимум придется проделать огромный объем инженерной работы.

Решение о том, кто должен отвечать за резервное копирование и восстановление приложений, может сопровождаться одними из самых жарких дебатов за все время существования организации. Можно сказать, что предоставление этих возможностей в виде одной из услуг платформы довольно заманчиво. С другой стороны, эта затея может потерять свою привлекательность, когда речь зайдет о нюансах, свойственных отдельным приложениям, например, попытка перезапуска приложения может оказаться неудачной и потребовать выполнения действия, о котором известно лишь разработчикам.

Одно из самых популярных средств резервного копирования данных как Kubernetes, так и приложений, — проект Velero (**https://velero.io**). Velero может создавать резервные копии объектов Kubernetes, если у вас есть желание их восстановить или перенести в другой кластер. Кроме того, Velero поддерживает создание копий то-

мов по расписанию. В ходе более подробного обсуждения этой темы вы узнаете, что планирование и администрирование копий *не* происходит автоматически. Зачастую нам доступны соответствующие механизмы, но мы сами должны построить на их основе процесс оркестрации. В заключение следует сказать, что Velero поддерживает хуки для резервного копирования и восстановления. Это позволяет выполнить необходимые нам команды перед созданием резервной копии или восстановлением ее содержимого. Например, некоторым приложениям нужно, чтобы перед резервным копированием был остановлен трафик или инициирован сброс данных на диск. Это можно осуществить с помощью хуков Velero.

Блочные устройства и хранение файлов/объектов

Определение типов хранилищ для наших приложений — ключевой фактор при выборе подходящей системы хранения и механизмов интеграции Kubernetes. Чаще всего в приложениях используются файловые хранилища, представляющие собой блочные устройства, поверх которых работает файловая система. Это позволяет приложениям осуществлять запись в файлы так, как мы это привыкли делать в любой ОС.

За файловой системой скрывается блочное устройство. Доступ к нему можно предоставлять в обход ФС, чтобы приложения могли взаимодействовать непосредственно с блоками данных. Файловые системы неизбежно повышают накладные расходы на операции записи. Если ваш рабочий сценарий требует прямого доступа к блочным устройствам, некоторые системы хранения это поддерживают, хотя в наши дни разработчики ПО редко об этом задумываются.

Существуют также хранилища объектного типа. Они отличаются от файловых отсутствием привычной иерархии. Разработчики могут взять неструктурированные данные, назначить им уникальный идентификатор, добавить какую-то метаинформацию и записать их в хранилище. Организации все чаще выбирают объектные хранилища от облачных провайдеров такие, как Amazon S3 (**https://aws.amazon.com/s3**), для размещения изображений, двоичных файлов и т. д. Этой тенденции способствует наличие полноценного API-интерфейса на основе HTTP и управления доступом. С объектными хранилищами чаще всего взаимодействуют прямо из приложений, при этом для аутентификации и обмена данными с провайдером предусмотрена специальная библиотека. Поскольку интерфейсы для работы с объектными хранилищами не так хорошо стандартизированы, они реже встречаются в виде сервисов платформы, с которыми могут прозрачно взаимодействовать приложения.

Временные данные

Продолжительность нахождения данных в хранилище может выходить за рамки жизненного цикла Pod'а, однако временные данные тоже имеют право на существование. По умолчанию данные, которые контейнеры записывают на файловую систему, попадают во временное хранилище. Если контейнер перезапустить, это хранилище будет утеряно. Для временных хранилищ, устойчивых к перезагрузкам, предусмотрен тип томов emptyDir (**https://oreil.ly/86zjA**). Он позволяет контейне-

рам не только сберечь данные между перезапусками, но и обмениваться файлами (при условии размещения в одном Pod'е).

Самый большой риск, связанный с временными данными, состоит в гарантии того, что ваши Pod'ы не занимают слишком много места в хранилище хоста. Несмотря на то, что 4 ГиБ для одного Pod'а не выглядит внушительно, но таких Pod'ов на одном узле может быть сотни, а в некоторых случаях и тысячи. Kubernetes позволяет ограничить совокупный объем временного хранилища, доступного Pod'ам в отдельно взятом пространстве имен. Связанная с этим конфигурация обсуждается в *главе 12*.

Выбор провайдера хранилища

Существует довольно широкий выбор провайдеров хранилищ, от продуктов с возможностью самостоятельного администрирования, таких как Ceph, до полностью управляемых систем наподобие Google Persistent Disk или Amazon Elastic Block Store. Разнообразие доступных вариантов выходит далеко за рамки этой книги. Тем не менее, мы советуем вам разобраться в возможностях систем хранения данных и понять, какие из этих возможностей можно легко интегрировать с Kubernetes. Это поможет вам получить представление о том, насколько хорошо одно решение удовлетворяет требованиям ваших приложений по сравнению с другим. Кроме того, если вы администрируете собственную систему хранения, предпочтение по возможности стоит отдавать тем провайдерам, с которыми вы уже имели дело. Внедрение Kubernetes параллельно с новой системой хранения принесет вашей организации много дополнительных трудностей в плане эксплуатации.

Механизмы для работы с хранилищами в Kubernetes

Kubernetes изначально предоставляет несколько механизмов для поддержки хранения данных в приложениях. Это составные элементы, с помощью которых мы будем предлагать многофункциональные хранилища. В этом разделе мы рассмотрим такие ресурсы, как PersistentVolume, PersistentVolumeClaim и StorageClass. В качестве примера будет показано выделение контейнерам заранее подготовленного хранилища.

Постоянные тома и заявки на выделение

Тома и заявки на выделение лежат в основе системы хранения данных Kubernetes. Они доступны в виде API-интерфейсов PersistentVolume (**https://oreil.ly/7_OAz**) и PersistentVolumeClaim (или PVC; **https://oreil.ly/PKtAr**). Ресурс PersistentVolume представляет том хранилища, известный платформе Kubernetes. Предположим, что администратор подготовил узел с быстрым внутренним хранилищем объемом 30 ГиБ. Также допустим, что путь к этому хранилищу имеет вид /mnt/fast-disk/pod-0. Чтобы сделать этот том доступным в Kubernetes, администратор может создать объект `PersistentVolume` (листинг 4.1).

Листинг 4.1

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0
spec:
  capacity:
    storage: 30Gi ❶
  volumeMode: Filesystem ❷
  accessModes:
  - ReadWriteOnce ❸
  storageClassName: local-storage ❹
  local:
    path: /mnt/fast-disk/pod-0
  nodeAffinity: ❺
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - test-w
```

❶ Объем хранилища, доступный в этом томе. Используется для определения возможности привязки к этому тому заявки на выделение.

❷ Определяет, является том блочным устройством (**https://oreil.ly/mrHwE**) или файловой системой.

❸ Задает режим доступа к тому: `ReadWriteOnce`, `ReadMany` или `ReadWriteMany`.

❹ Связывает этот том с классом хранилища, чтобы в дальнейшем привязать к этому тому заявку на выделение.

❺ Определяет, с каким узлом должен быть связан этот том.

Как видите, объект `PersistentVolume` содержит сведения, относящиеся к реализации тома. Для создания дополнительного уровня абстракции используется ресурс `PersistentVolumeClaim`, который привязывается к подходящему тому, когда его запрашивают. Чаще всего он определяется разработчиками приложения, которые добавляют его в свое пространство имен и обращаются к нему из своего Pod'а (листинг 4.2).

Листинг 4.2

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc0
```

```
spec:
  storageClassName: local-storage ❶
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 30Gi ❷
---
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: fast-disk
      persistentVolumeClaim:
        claimName: pvc0 ❸
  containers:
    - name: ml-processer
      image: ml-processer-image
      volumeMounts:
        - mountPath: "/var/lib/db"
          name: fast-disk
```

❶ Ищет том класса `local-storage` с режимом доступа `ReadWriteOnce`.

❷ Привязывается к тому размером больше или равным 30 ГиБ (`30Gi`).

❸ Объявляет этот Pod потребителем заявки `PersistentVolumeClaim`.

В зависимости от параметра `nodeAffinity` заявки `PersistentVolumeClaim`, выполнение Pod'а будет автоматически запланировано на хосте, где доступен этот том. От разработчика больше не требуется никакой дополнительной конфигурации подобия.

Сказанное является иллюстрацией того, насколько вовлеченными должны быть администраторы в процесс предоставления хранилища разработчикам. Мы называем этот поход статическим выделением. Если его как следует автоматизировать, он может служить вполне практичным способом предоставления быстрых дисков Pod'ам на определенных хостах. Например, в кластере можно развернуть компонент Local Persistence Volume Static Provisioner (**https://oreil.ly/YiQ0G**), который будет распознавать заранее выделенные хранилища и автоматически делать их доступными в качестве экземпляров объекта `PersistentVolume`. Он также имеет некоторые функции управления жизненным циклом, такие как удаление данных во время уничтожения объекта `PersistentVolumeClaim`.



Существует несколько механизмов для локального хранения данных, которые могут быть чреваты проблемами. Например, предоставление разработчикам доступа к hostPath (**https://oreil.ly/PAU8Y**) вместо выделения локального хранилища может показаться разумным решением, поскольку появляется возможность указать путь на хосте для привязки тома без `PersistentVolume` и `PersistentVolumeClaim`.

Но это может стать серьезной угрозой безопасности, так как у разработчиков появляется возможность привязываться к директориям хоста, что может иметь отрицательные последствия как для него, так и для других Pod-оболочек. Если вам хочется предоставить разработчикам временное хранилище, устойчивое к перезапуску Pod'ов, но не к их удалению или перемещению на другой узел, можете использовать EmptyDir (**https://oreil.ly/mPwBg**). Это позволит вам выделить в файловой системе хранилище, управляемое с помощью Kube и прозрачное для Pod'ов.

Классы хранилищ

Во многих ситуациях не реалистично ожидать, что узлы будут подготовлены заранее, вместе с дисками и томами. Такие случаи нередко оправдывают динамическое выделение, когда том может становиться доступным в зависимости от потребностей, описанных в нашей заявке. В рамках этой модели мы можем предоставить нашим разработчикам классы хранилищ, которые определяются с помощью API-интерфейса StorageClass (**https://oreil.ly/MoG_T**). Если предположить, что ваш кластер работает в AWS, и вы хотите динамически предоставлять Pod'ам тома EBS, это можно сделать путем добавления следующего ресурса `StorageClass` (листинг 4.3).

Листинг 4.3

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-standard ❶
  annotations:
    storageclass.kubernetes.io/is-default-class: true ❷
provisioner: kubernetes.io/aws-ebs ❸
parameters: ❹
  type: io2
  iopsPerGB: "17"
  fsType: ext4
```

❶ К объекту `StorageClass` можно обращаться по имени из заявок на выделение ресурсов.

❷ Делает этот экземпляр `StorageClass` классом по умолчанию. Он будет использоваться, если в заявке не указан класс.

❸ Задействует средство выделения ресурсов `aws-ebs` для создания томов на основе заявок.

❹ Конфигурация, которая относится к определенному провайдеру и описывает порядок выделения томов.

Вы можете предложить разработчикам широкий выбор систем хранения, сделав доступными несколько классов `StorageClass`. Это в том числе подразумевает поддержку нескольких провайдеров в одном кластере, например, Ceph вместе с VMware vSAN. В качестве альтернативы можно предложить разные уровни храни-

лиц от одного и того же провайдера. Скажем, хранилища можно разделить на дешевые и дорогие. К сожалению, у Kubernetes нет гибких механизмов, способных ограничивать разработчиков в выборе тех или иных классов. Но эти проверки можно реализовать в рамках контроля допуска, как это будет показано в *главе 8*.

Kubernetes имеет поддержку разнообразных провайдеров, включая AWS EBS, Glusterfs, GCE PD, Ceph RBD и многие другие. Так сложилось, что изначально они были размещены в иерархии основного проекта. Из-за этого провайдерам хранилищ приходилось реализовывать свою логику прямо в ядре Kubernetes, после чего этот код распределялся по соответствующим компонентам плоскости управления.

Эта модель имела несколько недостатков. Например, провайдер хранилища нельзя было разрабатывать за пределами основного проекта. Все изменения приходилось привязывать к выпуску Kubernetes. К тому же, каждая копия Kubernetes поставлялась вместе с лишним кодом. Например, кластеры, которые работали в AWS, содержали код провайдеров для взаимодействия с дисками GCE для постоянного хранения. Очень быстро стало понятно, что интеграцию с провайдерами было бы крайне полезно вынести за пределы проекта и объявить встроенную функциональность устаревшей. Вначале эту проблему пытались решить с помощью внешней спецификации FlexVolume (**https://oreil.ly/YnnCq**). Однако ее развитие было приостановлено в пользу интерфейса CSI, о котором пойдет речь дальше.

CSI

Интерфейс хранилищ для контейнеров (англ. Container Storage Interface или CSI) позволяет предоставлять блочные и файловые хранилища нашим приложениям. Его реализации называют драйверами, они знают, как взаимодействовать с провайдерами хранилищ. В число этих провайдеров входят как облачные сервисы наподобие Google Persistent Disks (**https://cloud.google.com/persistent-disk**), так и системы хранения для самостоятельного развертывания и администрирования (такие как Ceph; **https://ceph.io**). Драйверы реализуются провайдерами хранилищ в рамках проектов, находящихся за пределами кодовой базы Kubernetes. Их разработкой можно заниматься совершенно отдельно от кластера, в котором они развернуты.

Если не вдаваться в подробности, то реализации CSI состоят из двух подключаемых модулей: для контроллера и узла. У разработчиков драйвера есть большая свобода действий при реализации этих компонентов. Обычно модули для контроллера и узла объединяются в один исполняемый файл, который переключается в тот или иной режим с помощью переменной окружения наподобие `X_CSI_MODE`. От драйвера требуется только две вещи — чтобы он зарегистрировался с помощью kubelet и содержал реализацию конечных точек, предусмотренных в спецификации CSI.

Сервис контроллера управляет созданием и удалением томов в провайдере хранилищ. Эту функциональность можно (при желании) дополнить такими возможностями, как расширение томов и создание их копий. Сервис узла отвечает за подготовку томов к использованию Pod'ами, размещенными на узле. Зачастую это выражается в настройке точек подключения и предоставлении информации о томах. Оба сервиса также реализуют механизмы идентификации, которые возвращают сведе-

ния о подключаемом модуле, его возможностях и текущей работоспособности. На рис. 4.1 представлена архитектура кластера, в котором развернуты эти компоненты.

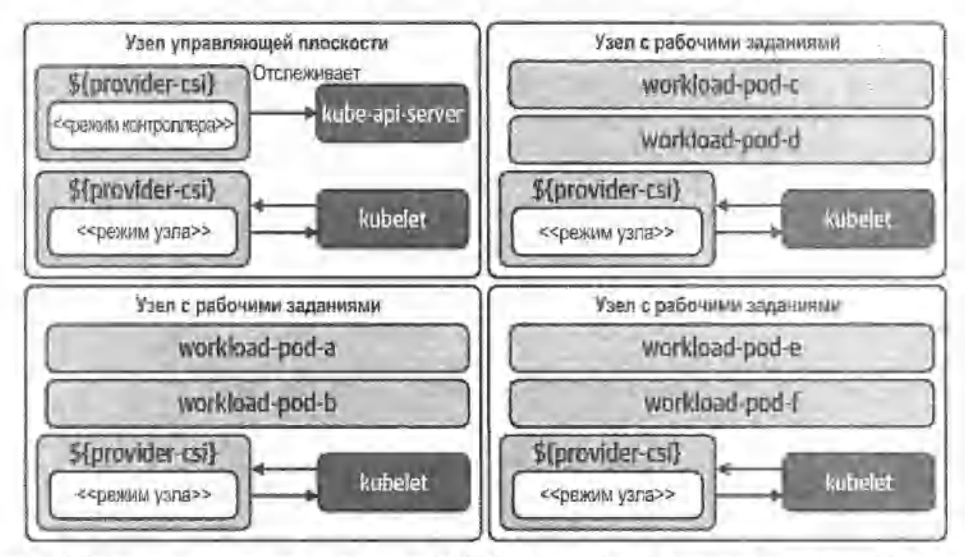


Рис. 4.1. Кластер с подключаемым модулем CSI. Драйвер работает в режимах узла и контроллера. Контроллер, как правило, имеет вид объекта Deployment. Сервис узла развертывается в качестве объекта DaemonSet, который размещает по одной Pod-оболочке на каждом хосте

Давайте подробнее рассмотрим эти два компонента — контроллер и узел.

Контроллер CSI

Сервис контроллера CSI предоставляет API-интерфейс для работы с томами в системе постоянного хранения. Плоскость управления Kubernetes *не* взаимодействует с ним напрямую. Вместо этого контроллеры, поддержкой которых занимается сообщество, реагируют на события Kubernetes и преобразовывают их в такие инструкции CSI, как `CreateVolumeRequest` при создании заявки `PersistentVolumeClaim`. Поскольку сервисы контроллеров CSI предоставляет свои API-интерфейсы в виде UNIX-сокетов, сами контроллеры обычно развертываются вместе с этим сервисом по принципу "прицепа". Существует несколько внешних контроллеров с разным поведением:

- *external-provisioner* — запрашивает у драйвера CSI создание тома при создании заявки `PersistentVolumeClaim`. Как только в провайдере хранилища появится новый том, это средство выделения ресурсов создаст в Kubernetes объект `PersistentVolume`.
- *external-attacher* — следит за объектами `VolumeAttachment`, которые объявляют о присоединении или отсоединении тома от узла. Отправляет драйверу CSI запросы на присоединение или отсоединение.
- *external-resizer* — распознает изменения размера хранилища в заявках `PersistentVolumeClaim`. Отправляет драйверу CSI запросы на расширение томов.

- *external-snapshotter* — при создании объектов `VolumeSnapshotContent` драйверу направляются запросы на создание копий.



При реализации подключаемых модулей для CSI разработчикам необязательно использовать перечисленные контроллеры. Однако их применение поощряется, чтобы предотвратить дублирование логики в каждом подключаемом модуле.

Узел CSI

Подключаемый модуль Node обычно содержит тот же код драйвера, что и контроллер. Но, поскольку он работает в "режиме узла", его основными обязанностями являются такие задачи, как подключение присоединенных томов, определение их файловой системы и подключение томов к Pod'ам. Запросы на выполнение этих операций выполняются с использованием kubelet. Вместе с драйвером в Pod, как правило, размещаются следующие компоненты:

- *node-driver-registrar* — отправляет агенту kubelet запрос на регистрацию (**https://oreil.ly/kmkJh**), чтобы уведомить его о драйвере CSI;
- *liveness-probe* — сообщает о работоспособности драйвера CSI.

Реализация хранилища в виде сервиса

Мы обсудили ключевые вопросы, касающиеся хранения данных в приложениях, соответствующих механизмов, доступных в Kubernetes, и интеграции драйверов с использованием CSI. Теперь давайте соберем все эти идеи воедино и рассмотрим реализацию, которая предоставляет разработчикам хранилище в виде сервиса. Мы хотим сделать так, чтобы запрашивать хранилище и открывать к нему доступ для приложений можно было декларативным образом. Желательно делать это динамически, без обращения к администратору с просьбой выделить и присоединить тот или иной том. Нам бы хотелось, чтобы этот процесс происходил автоматически в зависимости от потребностей.

Для этой реализации мы будем использовать Amazon Web Services (AWS). Данный пример интегрируется с эластичным хранилищем данных AWS (**https://oreil.ly/I4VVw**). Этот материал будет полезен, даже если вы пользуетесь услугами другого провайдера! Мы выбрали AWS, чтобы показать конкретный пример, как все эти компоненты работают вместе.

Теперь поговорим о подготовке интеграции/драйвера, предоставлении разработчикам разных вариантов хранения данных, обращении к хранилищу из приложений, изменении размера томов и создании их копий.

Установка компонентов CSI

Этап подготовки довольно прост и состоит из двух шагов:

1. Настройка доступа к провайдеру.
2. Развертывание компонентов драйвера в кластере.

Провайдер (в нашем случае AWS) требует, чтобы драйвер идентифицировался, что позволяет выдать ему подходящие права доступа. В данном примере это можно сделать тремя способами. Первый — обновить профиль инстансов (**https://oreil.ly/fGWYd**) для узлов Kubernetes. Благодаря этому нам не придется беспокоиться об учетных данных на уровне Kubernetes, так как приложениям, способным обращаться к API-интерфейсу AWS, выдаются универсальные привилегии. Второй и, наверное, самый безопасный вариант состоит в использовании сервиса идентификации, который будет выдавать отдельным приложениям права доступа IAM. В качестве примера можно привести проект kiam (**https://github.com/uswitch/kiam**). Этот подход рассматривается в *главе 10*. Последний способ заключается в том, чтобы добавить учетные данные в объект `Secret`, который затем подключается к драйверу CSI. В этом случае объект `Secret` выглядит так:

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-secret
  namespace: kube-system
stringData:
  key_id: "AKIAWJQHICPELCJVKYNU"
  access_key: "jqWi1ut4KyrAHADIOrhH2Pd/vXpgqA9OZ3bCZ"
```



Эта учетная запись получит право манипулировать внутренней системой хранения. Доступ к данному ресурсу должен тщательно контролироваться. Больше об этом можно узнать в *главе 7*.

Подготовив эту конфигурацию, мы можем приступить к установке компонентов CSI. Сначала устанавливается контроллер в виде объекта `Deployment`. При существовании нескольких реплик он будет использовать алгоритм выбора лидера для определения того, какой инстанс должен быть активным. Затем устанавливается подключаемый модуль узла, который имеет вид объекта `DaemonSet`, размещающего Pod'ы на каждом узле. После инициализации экземпляры подключаемого модуля узла регистрируются, обращаясь к своим агентам kubelet, а те в свою очередь сообщают платформе об узле с поддержкой CSI, создавая объект `CSINode` на каждом узле Kubernetes. В кластере из трех узлов вывод выглядит так:

```
$ kubectl get csinode

NAME                                        DRIVERS   AGE
ip-10-0-0-205.us-west-2.compute.internal    1         97m
ip-10-0-0-224.us-west-2.compute.internal    1         79m
ip-10-0-0-236.us-west-2.compute.internal    1         98m
```

Мы видим три узла, в каждом из которых зарегистрировано по одному драйверу. Если заглянуть в YAML-файл одного из объектов `CSINode`, можно увидеть код, приведенный в листинге 4.4.

Листинг 4.4

```
apiVersion: storage.k8s.io/v1
kind: CSINode
metadata:
  name: ip-10-0-0-205.us-west-2.compute.internal
spec:
  drivers:
    - allocatable:
        count: 25 ❶
      name: ebs.csi.aws.com
      nodeID: i-0284ac0df4da1d584
      topologyKeys:
        - topology.ebs.csi.aws.com/zone ❷
```

❶ Максимальное число томов, допустимое на этом узле.

❷ Когда узел выбирается для рабочего задания, это значение передается в запросе CreateVolumeRequest, чтобы драйвер знал, *где* нужно создать том. Это важно в случаях, когда узлы кластера не имеют доступа к одному и тому же хранилищу. Например, если взять AWS, то при планировании развертывания Pod'а его том должен быть создан в той же зоне доступности.

В дополнение к этому драйвер официально регистрируется в кластере. Подробности об этом можно найти в манифесте объекта `CSIDriver` (листинг 4.5).

Листинг 4.5

```
apiVersion: storage.k8s.io/v1
kind: CSIDriver
metadata:
  name: aws-ebs-csi-driver ❶
  labels:
    app.kubernetes.io/name: aws-ebs-csi-driver
spec:
  attachRequired: true ❷
  podInfoOnMount: false ❸
  volumeLifecycleModes:
  - Persistent ❹
```

❶ Этот драйвер представлен именем провайдера, которое привязывается к классу хранилищ, предлагаемых пользователю платформы.

❷ Говорит о том, что перед подключением томов должна быть завершена операция присоединения.

❸ При конфигурации подключения метаданные Pod'а не нужно передавать в виде контекста.

❹ Модель по умолчанию для выделения постоянных томов. Поддержку встроенных томов (**https://oreil.ly/Z_pDY**) можно включить, присвоив этому параметру

значение `Ephemeral`. В этом режиме хранилище должно существовать не дольше, чем сам Pod.

Параметры и объекты, рассмотренные до сих пор, являются элементами нашего процесса начальной конфигурации. Объект `CSIDriver` облегчает обнаружение информации о драйвере и входит в его пакет развертывания. Объект `CSINode` управляется агентом kubelet. Универсальное средство регистрации добавляется в Pod подключаемого модуля узла, получает сведения от драйвера CSI и регистрирует этот драйвер с помощью kubelet. После этого kubelet сообщает платформе о том, сколько драйверов CSI доступно на каждом хосте. Этот процесс начальной конфигурации продемонстрирован на рис. 4.2.

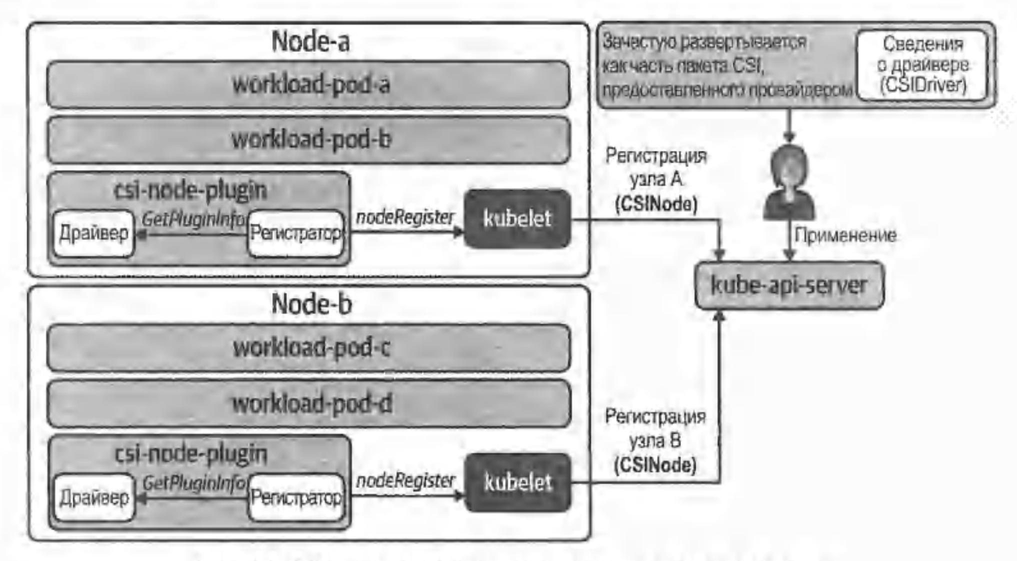


Рис. 4.2. Объект `CSIDriver` развертывается как часть пакета, а подключаемый модуль узла регистрируется с помощью kubelet. В результате появляется возможность создавать/администрировать объекты `CSINode`

Предоставление разных вариантов хранилищ

Чтобы предоставить разработчикам разные варианты хранилищ, нужно создать объекты `StorageClass`. В следующем примере мы хотим сделать доступными хранилища двух типов. Первый тип дает доступ к дешевому диску, который приложения могут использовать для постоянного хранения данных. Многим приложениям не нужны накопители SSD, так как они сохраняют небольшое число файлов, что не требует быстрого ввода/вывода. Таким образом, дешевый жесткий диск (HDD) будет вариантом по умолчанию. Но мы также хотим предложить более быстрый твердотельный накопитель (SSD) с возможностью погигабайтной настройки IOPS (**https://ru.wikipedia.org/wiki/IOPS**). Наши предложения показаны в табл. 4.1; цены соответствуют тарифам AWS на момент написания этих строк.

Таблица 4.1. Варианты хранилищ

Название предложения	Тип хранилища	Максимальная пропускная способность тома	Стоимость в AWS
default-block	HDD (оптимизированный)	40...90 Мб/с	0,045 долл. за Гб в месяц
performance-block	SSD (io1)	~1 000 Мб/с	0,125 долл за Гб в месяц + 0,065 долл. в месяц за каждое выделение IOPS

Чтобы предоставить эти варианты, мы создадим для каждого из них класс хранилищ. Внутри каждого класса есть поле `parameters`. Именно с его помощью мы можем указать конфигурацию, которая соответствует возможностям, перечисленным в табл. 4.1 (листинг 4.6).

Листинг 4.6

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: default-block ❶
  annotations:
    storageclass.kubernetes.io/is-default-class: "true" ❷
provisioner: ebs.csi.aws.com ❸
allowVolumeExpansion: true ❹
volumeBindingMode: WaitForFirstConsumer ❺
parameters:
  type: st1 ❻
---
kind: StorageClass ❼
apiVersion: storage.k8s.io/v1
metadata:
  name: performance-block
provisioner: ebs.csi.aws.com
parameters:
  type: io1
  iopsPerGB: "20"
```

❶ Название типа хранилища, который мы предлагаем пользователям платформы. На него будут ссылаться заявки `PeristentVolumeClaim`.

❷ Это делает тип вариантом по умолчанию. Если заявка `PeristentVolumeClaim` создается *без* указания `StorageClass`, то используется `default-block`.

❸ Определяет драйвер CSI, к которому нужно привязываться.

❹ Возможность увеличения размера тома путем внесения изменений в `PersistentVolumeClaim`.

❺ Том не выделяется, пока Pod не обработает `PersistentVolumeClaim`. Это гарантирует, что том будет создан в одной зоне доступности с Pod, развертывание которой мы планируем. Также не дает утраченным заявкам создавать в AWS тома, за которые вам придется платить.

❻ Определяет тип хранилища, которое должен предоставить драйвер, чтобы удовлетворить заявки.

❼ Второй класс, рассчитанный на высокую производительность.

Использование хранилища

Мы подготовили все описанные компоненты и готовы к тому, чтобы предоставить пользователям доступ к разным классам хранения данных. Для начала посмотрим, как разработчик запрашивает хранилище. Затем обсудим механизмы, которые делают возможным этот процесс. Давайте посмотрим на то, что получает разработчик, когда запрашивает список доступных объектов `StorageClass` (листинг 4.7).

Листинг 4.7

```
$ kubectl get storageclasses.storage.k8s.io

NAME                     PROVISIONER         RECLAIMPOLICY   VOLUMEBINDINGMODE
default-block (default)  ebs.csi.aws.com     Delete          Immediate
performance-block        ebs.csi.aws.com     Delete          WaitForFirstConsumer

ALLOWVOLUMEEXPANSION
true
true
```



Позволив разработчикам создавать объекты `PersistentVolumeClaim`, мы разрешим им ссылаться на *любой* класс `StorageClass`. Если такой подход проблематичен, вам, возможно, стоит реализовать контроль допуска, чтобы оценивать уместность запросов. Эта тема рассматривается в *главе 8*.

Предположим, разработчик хочет предоставить приложениям на выбор дешевые HDD и высокопроизводительные SSD. В этом случае нужно создать два запроса `PersistentVolumeClaim`. Назовем их `pvc0` и, соответственно, `pvc1` (листинг 4.8).

Листинг 4.8

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc0 ❶
spec:
  resources:
    requests:
      storage: 11Gi
```

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc1
spec:
  resources:
    requests:
      storage: 14Gi
  storageClassName: performance-block ❷
```

❶ Будет использоваться класс хранилищ `default-block` и другие параметры по умолчанию, такие как RWO и тип хранилищ на основе файловой системы.

❷ Делает так, что у драйвера запрашивается `performance-block`, а не `default-block`.

Эти два запроса будут выделять разные ресурсы в зависимости от параметров `StorageClass`. Высокопроизводительное хранилище (из `pvc1`) создается в виде неприсоединенного тома в AWS. Этот том готов к использованию, и его можно быстро присоединить. Хранилище по умолчанию (из `pvc0`) будет находиться в состоянии `Pending`, в котором кластер ждет, когда Pod обработает PVC, и затем выделяет ресурсы в AWS. Когда Pod, наконец, обработает запрос, процесс выделения будет более трудоемким, однако вам не придется платить за неиспользуемое хранилище! Взаимосвязь между запросом в Kubernetes и томом в AWS представлена на рис. 4.3.



Рис. 4.3. `pv1` выделяется в виде тома в AWS, а CSIVolumeName используется для простоты сопоставления; том для `pv0` создается только после того, как на него сошлется Pod-оболочка

Теперь представим, что разработчик создал два Pod'а: один ссылается на `pv0`, а другой на `pv1`. Когда развертывание каждого Pod'а будет запланировано, к соответствующему узлу будет присоединен подходящий том. В случае с `pv0` том предварительно создается в AWS. Дальше определяется файловая система, и к контейнеру подключается хранилище. Поскольку речь идет о постоянных томах, мы фактически создали модель, в которой том может быть перенесен на другой узел вместе с Pod'ом. Весь процесс того, как мы выделяем хранилище по запросу, показан на рис. 4.4.

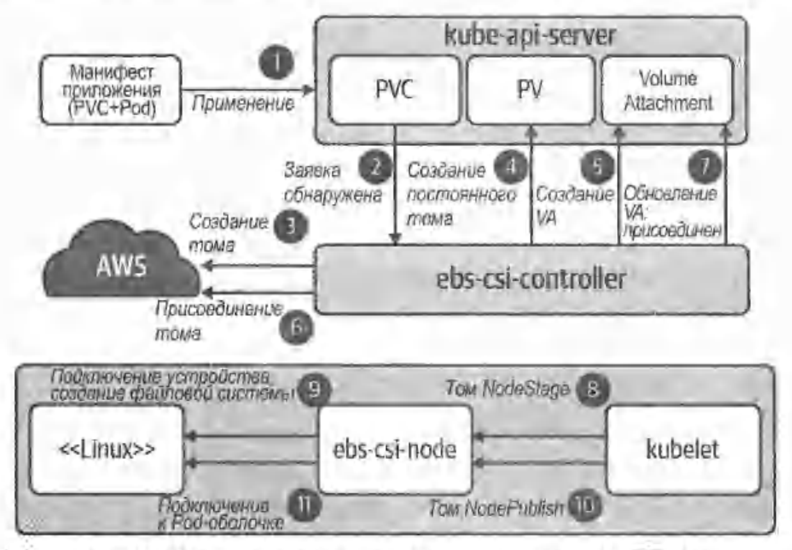


Рис. 4.4. Весь процесс взаимодействия драйвера и Kubernetes в ходе выделения хранилища по запросу



События особенно полезны для отладки взаимодействия хранилища и CSI. Поскольку выделение, присоединение и подключение выполняются для удовлетворения заявки PVC, события, происходящие с этими объектами, следует считать отчетами от компонентов о проделанной работе. Команда `kubectl describe -n $NAMESPACE pvc $PVC_NAME` позволяет легко просматривать эти события.

Изменение размера

Команда `aws-ebs-csi-driver` поддерживает изменение размера. В большинстве реализаций CSI обнаружением изменений в объектах `PersistentVolumeClaim` занимается контроллер `external-resizer`. Событие изменения размера направляется драйверу, который расширяет том. В данном случае драйвер, работающий в подключаемом модуле контроллера, выполняет операцию расширения с помощью API-интерфейса AWS EBS.

Контейнер может пользоваться новым пространством *не* сразу после расширения тома в EBS. Дело в том, что файловая система по-прежнему имеет исходный размер. Нам нужно будет подождать, пока экземпляр драйвера в подключаемом модуле узла ее не расширит. Все это можно проделать *без* выключения Pod'а. Расширение файловой системы можно наблюдать в следующих журнальных записях, сгенерированных драйвером CSI из подключаемого модуля узла:

```
mount_linux.go: Attempting to determine if disk "/dev/nvme1n1" is formatted
using blkid with args: ([-p -s TYPE -s PTTYPE -o export /dev/nvme1n1])

mount_linux.go: Output: "DEVNAME=/dev/nvme1n1\nTYPE=ext4\n", err: <nil>

resizefs_linux.go: ResizeFS.Resize - Expanding mounted volume /dev/nvme1n1

resizefs_linux.go: Device /dev/nvme1n1 resized successfully
```



Kubernetes не поддерживает уменьшение размера с помощью поля `storage` в PVC. Если у драйвера CSI нет решения этой проблемы, вы, скорее всего, сможете уменьшить том только путем его повторного создания. Имейте это в виду, когда расширяете свои тома.

Копии (snapshots)

Для периодического резервного копирования данных тома, которые используются контейнерами, предусмотрена возможность создания копий. Эта функциональность зачастую разделена на два контроллера, которые отвечают за разные определения пользовательских ресурсов (англ. Custom Resource Definitions или CRD): VolumeSnapshot и VolumeContentSnapshot. Если не вдаваться в подробности, то VolumeSnapshot управляет жизненным циклом томов. Управление копиями VolumeContentSnapshot осуществляется контроллером external-snapshotter с использованием этих объектов. Этот контроллер обычно развертывается по принципу "прицепа" в подключаемом модуле контроллера CSI и направляет запросы драйверу.



На момент написания этих строк упомянутые ранее объекты реализованы в виде CRD, а не как часть основного API-интерфейса Kubernetes. В связи с этим драйвер CSI или дистрибутив Kubernetes должен развернуть эти определения заранее.

Подобно тому, как классы `StorageClass` предлагают хранилища, копии можно сделать доступными путем создания класса `Snapshot`. Этот класс представлен в листинге 4.9 в формате YAML.

Листинг 4.9

```
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshotClass
metadata:
  name: default-snapshots
driver: ebs.csi.aws.com ❶
deletionPolicy: Delete ❷
```

❶ Какому драйверу делегировать запрос копии.

❷ Следует ли `VolumeSnapshotContent` удалять вместе с `VolumeSnapshot`. Это фактически делает возможным удаление самого тома (в зависимости от того, поддерживает ли это провайдер).

Контроллер `VolumeSnapshot` можно создать в пространстве имен приложения и `PersistentVolumeClaim` (листинг 4.10).

Листинг 4.10

```
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshot
metadata:
  name: snap1
```

```
spec:
  volumeSnapshotClassName: default-snapshots ❶
  source:
    persistentVolumeClaimName: pvc0 ❷
```

❶ Класс, который будет использовать драйвер.

❷ Заявка на выделение тома, определяющая том, копию которого нужно сделать.

Существование этого объекта сигнализирует о необходимости создания `VolumeSnapshotContent`, действующего на уровне всего кластера. Обнаружение `VolumeSnapshotContent` приведет к отправке запроса на создание копии, и драйвер его удовлетворит путем взаимодействия с AWS EBS. После этого `VolumeSnapshot` поменяет свое состояние на `ReadyToUse`. На рис. 4.5 продемонстрированы отношения между различными объектами.

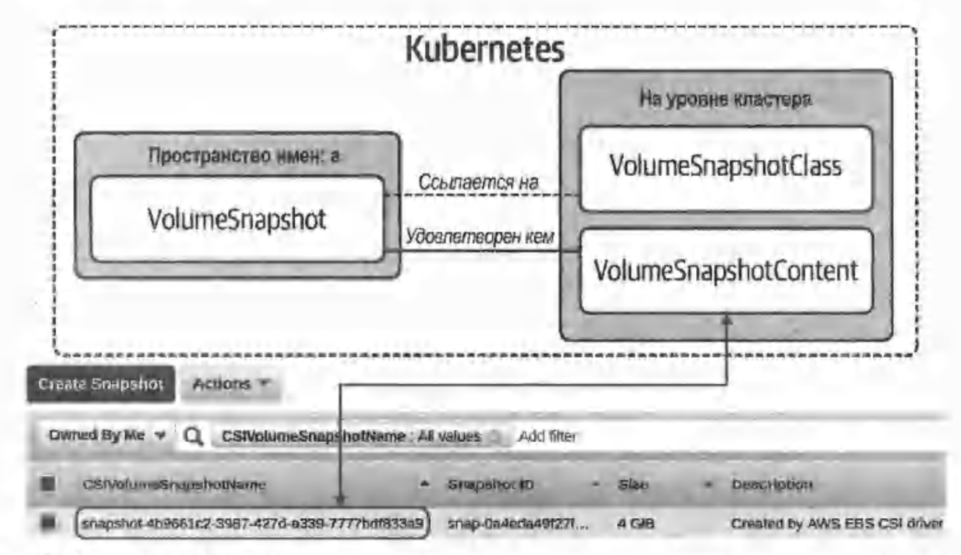


Рис. 4.5. Различные объекты и отношения между ними, из которых состоит процесс создания копии

Теперь, получив копию, мы можем обсудить потенциальную потерю данных, вызванную случайным удалением исходного тома, его сбоем или случайным удалением `PersistentVolumeClaim`. Во всех этих случаях данные могут быть восстановлены. Для этого нужно создать новую заявку `PersistentVolumeClaim` с полем `spec.dataSource`. В этом поле можно сослаться на объект `VolumeSnapshot`, который перенесет данные в новую заявку. Следующий манифест восстанавливается из ранее созданной копии (листинг 4.10).

Листинг 4.10

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-reclaim
```

```
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: default-block
  resources:
    requests:
      storage: 600Gi
  dataSource:
    name: snap1 ❶
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
```

❶ Экземпляр `VolumeSnapshot`, который ссылается на копию EBS, чтобы наполнить данными новую заявку PVC.

После повторного создания Pod'a, которая ссылается на эту новую заявку, контейнер вернется в состояние, зафиксированное в последней копии! Теперь мы имеем в своем распоряжении все механизмы для создания надежного средства резервного копирования и восстановления. Регулярное создание резервных копий объектов Kubernetes и томов с данными можно осуществлять путем генерации копий по расписанию с помощью CronJob, написания собственного контроллера или использования таких инструментов, как Velero (**https://velero.io**).

Резюме

В этой главе мы рассмотрели разнообразные темы, связанные с хранением данных в контейнерах. Для принятия обоснованных технических решений необходимо хорошо понимать требования вашего приложения. Также нужно убедиться в том, что провайдер хранилища способен удовлетворить эти требования, и что у нас есть знания и опыт, необходимые для работы с ним (если он действительно нужен). И, наконец, мы должны организовать взаимодействие между оркестратором и системой хранения данных так, чтобы для выбора подходящего хранилища разработчикам не нужно было разбираться в этой системе.

ГЛАВА 5

Сетевое взаимодействие между Pod'ами

С самого момента появления компьютерных сетей мы задаемся вопросами о том, как организовать взаимодействие между разными хостами. В число этих вопросов входят: обращение к хостам по уникальным адресам, маршрутизация пакетов по сетям и распространение известных маршрутов. Уже больше десяти лет для решения подобных задач в наших все более динамичных окружениях используют программно-определяемые сети (англ. Software-Defined Networks или SDN), и их популярность стремительно растет. Вы и сами, скорее всего, являетесь их пользователем, и неважно, с какими технологиями вы работаете — VMware NSX в центре обработки данных или Amazon VPC в облаке.

Эти принципы и рекомендации применимы, в том числе и к Kubernetes. И хотя в нашем случае речь идет о Pod'ах, а не о хостах, необходимо обеспечить адресацию и маршрутизацию для наших приложений. Кроме того, учитывая, что Pod'ы — это программное обеспечение, которое выполняется на наших хостах, сети, которые мы организовываем, в большинстве случаев являются полностью программно-определенными.

В этой главе мы опишем концепцию сетей, состоящих из Pod-оболочек. Для начала будут рассмотрены некоторые ключевые понятия, которые следует понимать и учитывать при реализации Pod-сетей. Мы поговорим о CNI (Container Networking Interface — интерфейс управления сетью контейнеров; **https://github.com/containernetworking/cni**), который позволит вам выбрать реализацию сети с учетом ваших требований. Напоследок мы обсудим такие подключаемые модули, как Calico и Cilium, распространенные в экосистеме Kubernetes, чтобы добавить конкретики в процесс выбора подходящей конфигурации. По прочтении этой главы вы будете лучше готовы к принятию решений о сетевых решениях и параметрах вашей платформы приложений.



Сеть сама по себе является обширной темой. Мы намереваемся дать вам ровно столько знаний, сколько необходимо для принятия обоснованных решений о вашей Pod-сети. Если у вас нет опыта работы с сетями, настоятельно советуем обсудить эти концепции с вашими сетевыми администраторами. Kubernetes не избавляет вашу организацию от необходимости в специалистах по сетям!

Аспекты, связанные с сетью

Прежде чем переходить к деталям реализации Pod-сетей, нужно сначала рассмотреть несколько ключевых аспектов, включая следующие:

- управление IP-адресами (англ. IP Address Management или IPAM);
- протоколы маршрутизации;
- инкапсуляция и туннелирование;
- маршрутизируемость приложений;
- IPv4 и IPv6;
- шифрование трафика;
- сетевая политика.

Ориентируясь в этих областях, вы сможете приступить к выбору подходящего сетевого решения для своей платформы.

Управление IP-адресами

Чтобы Pod'ы могли принимать и отправлять сетевой трафик, мы должны предусмотреть для них уникальные адреса. В Kubernetes каждому Pod'у назначается IP-адрес. Он может быть внутрикластерным или доступным снаружи. Наличие у каждого Pod'а собственного адреса упрощает сетевую модель, так как нам не придется беспокоиться о конфликтующих портах в системах с разделяемыми IP-адресами. Однако назначение каждому Pod'у собственного IP-адреса имеет свои трудности.

Pod лучше всего воспринимать как временный компонент. В частности, их нередко перезапускают и разворачивают на других узлах в зависимости от потребностей кластера или в ответ на сбой системы. Это требует быстрого выделения IP-адресов, а процесс управления их пулом должен быть эффективным. Последний часто называют IPAM (IP Address Management — управление IP-адресами; **https://oreil.ly/eWJki**), и его применение не ограничивается Kubernetes. При более детальном рассмотрении методов организации сетей из контейнеров мы исследуем разнообразные способы реализации IPAM.



Тот факт, что IP-адреса являются временными, вызывает проблемы с некоторыми устаревшими приложениями, которые, например, пытаются привязаться к определенному IP-адресу и рассчитывают на то, что он будет оставаться неизменным. Некоторые реализации сети контейнеров (о которых речь пойдет позже в этой главе) позволяют явно резервировать IP-адреса для определенных приложений. Но мы советуем использовать такую модель только в случаях, когда это необходимо. Существует множество функциональных механизмов DNS и обнаружения сервисов, с помощью которых приложения могут решить эту проблему. Примеры ищите в *главе 6*.

Реализация IPAM зависит от того, какой подключаемый модуль CNI вы выберете. У этих модулей есть несколько общих свойств, связанных с Pod'ами. Прежде всего, при создании кластеров можно указать CIDR (Classless Inter-Domain Routing — бесклассовая адресация; **https://oreil.ly/honRv**) Pod-сети. То, как именно это дела-

ется, зависит от метода начальной конфигурации Kubernetes. В случае с kubeadm можно передать флаг так:

```
kubeadm init --pod-network-cidr 10.30.0.0/16
```

Эта команда фактически устанавливает флаг `--cluster-cidr` для kube-controller-manager. Затем Kubernetes выделит часть этого пула для каждого узла. По умолчанию каждый узел получает подсеть `/24`. Но это можно изменить с помощью флага `--node-cidrmask-size-ipv4` и/или `--node-cidr-mask-size-ipv6` для kube-controller-manager. В листинге 5.1 приведен пример выделения адресов с помощью объекта `Node`.

Листинг 5.1

```
apiVersion: v1
kind: Node
metadata:
  labels:
    kubernetes.io/arch: amd64
    kubernetes.io/hostname: test
    kubernetes.io/os: linux
    manager: kubeadm
  name: master-0
spec:
  podCIDR: 10.30.0.0/24 ❶
  podCIDRs:
  - 10.30.0.0/24 ❷
```

❶ Это поле существует для совместимости. Его более новая разновидность, `podCIDRs`, имеет вид массива и позволяет выделять для каждого узла двойные диапазоны (IPv4 и IPv6 CIDR).

❷ Этот узел имеет диапазон IP-адресов 10.30.0.0 – 10.30.0.255. Таким образом, из 65 534 адресов в подсети кластера 10.30.0.0/16 Pod'ам доступно 254.

Будут эти значения использоваться в IPAM или нет, зависит от подключаемого модуля CNI. Например, Calico эти параметры распознает и соблюдает, а Cilium делает их соблюдение опциональным, предлагая по умолчанию управлять пулами IP-адресов отдельно от Kubernetes. Большинство реализаций CNI требуют, чтобы выбранный вами диапазон (CIDR) не пересекался с сетью хостов/узлов кластера. Хотя, если принять, что ваша Pod-сеть будет оставаться внутри кластера, ваш диапазон может пересекаться с внешним сетевым пространством. На рис. 5.1 представлены взаимосвязи между различными диапазонами IP-адресов и примеры их выделения.



Размер диапазона для Pod'ов кластера зачастую продиктован сетевой моделью. В большинстве развертываний Pod-сеть находится целиком внутри кластера. В связи с этим диапазон можно сделать очень большим с расчетом на будущий рост. Если же Pod'ы доступны из более крупной сети, то, чтобы не занимать адресное пространство, вам, возможно, придется обдумать этот вопрос более тщательно. Приблизительный размер можно получить, умножив число Pod'ов, развернутых на каждом узле, на число узлов, которое вы ожидаете в будущем. Число Pod'ов на узле настраивается с помощью kubelet и по умолчанию равно 110.

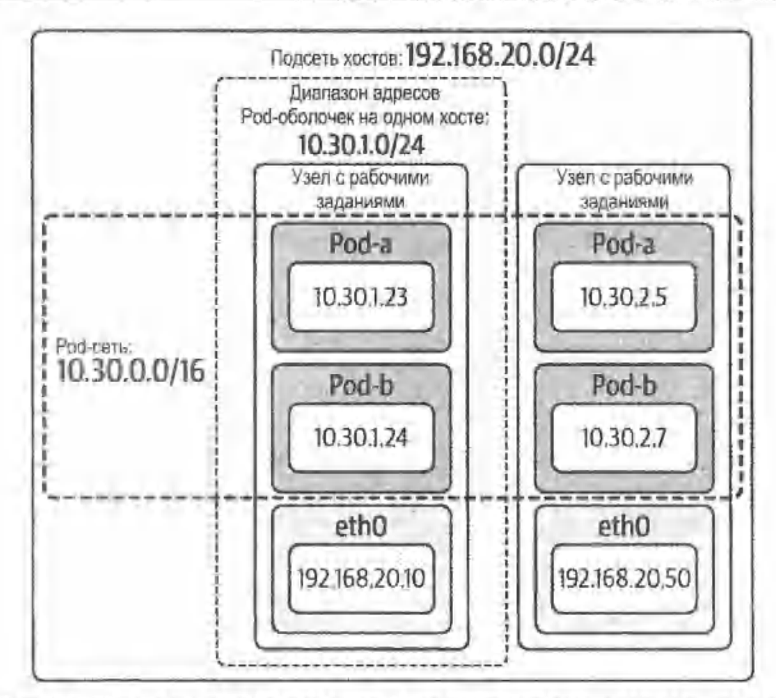


Рис. 5.1. Пространства IP-адресов и диапазоны, выделенные для сети хостов, Pod-сети и Pod-оболочек на каждом отдельном хосте

Протоколы маршрутизации

Назначив Pod'ам адреса, мы должны позаботиться о том, чтобы все участники сети понимали маршруты, которые ведут к ним и от них. Для этого существуют протоколы маршрутизации. Их можно считать разными методами распространения информации о входящих и исходящих маршрутах. Внедрение такого протокола зачастую делает возможной динамическую маршрутизацию в противовес настройке статических маршрутов (**https://oreil.ly/97En2**). Во многих случаях сеть "не знает", как маршрутизировать IP-адреса приложений, поэтому при отсутствии в Kubernetes инкапсуляции (о которой мы поговорим в следующем разделе) необходимо ориентироваться в многочисленных маршрутах.

Один из самых распространенных протоколов для распределения маршрутов между приложениями — BGP (Border Gateway Protocol — протокол граничного шлюза). Он используется в таких проектах, как Calico (**https://www.projectca lico.org**) и Kube-Router (**https://www.kube-router.io**). BGP не только делает возможным взаимодействие между маршрутами в кластере, но также позволяет обмениваться трафиком внутренним и внешним маршрутизаторам. Таким образом, механизмы внешней сети могут знать, как направлять трафик к IP-адресам Pod'ов. Сервис BGP работает внутри Pod'a Calico, который размещен на каждом узле. По мере того, как маршруты к приложениям становятся известны, Pod Calico вносит изменения в таблицу маршрутизации ядра, добавляя в нее маршруты к каждому приложению.

Это обеспечивает стандартную маршрутизацию по IP-адресам приложений, что может быть особенно удачным решением, если все они находятся в одном и том же сегменте L2. Это поведение продемонстрировано на рис. 5.2.

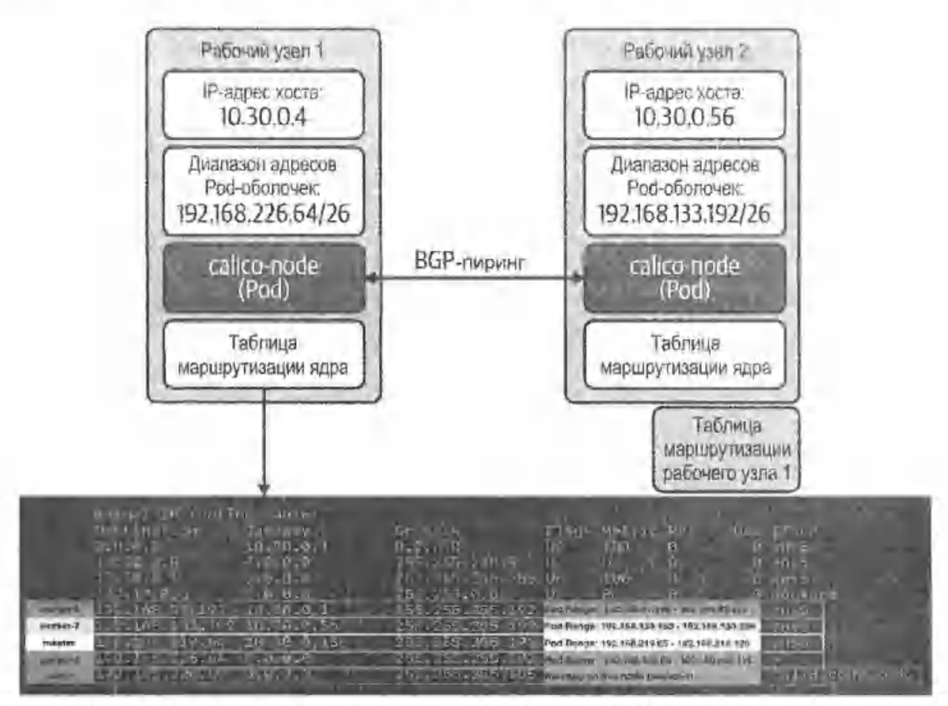


Рис. 5.2. Pod-оболочка Calico делится маршрутами посредством BGP-пиринга. Затем соответствующим образом программируется таблица маршрутизации ядра



Предоставление доступа к IP-адресам Pod'ов из более крупных сетей может казаться заманчивым на первый взгляд, но требует тщательного взвешивания. Более подробно это обсуждается в разделе "Маршрутизируемость приложений" данной главы.

Во многих окружениях стандартная маршрутизация IP-адресов приложений не представляется возможной. Кроме того, протоколы маршрутизации наподобие BGP иногда нельзя интегрировать в саму сеть, как в случае с сетями облачных провайдеров. Возьмем в качестве примера развертывание CNI, в котором мы хотим поддерживать стандартную маршрутизацию и обмениваться маршрутами по BGP. В окружении AWS эта затея может провалиться по двум причинам:

- *Включены проверки источника/назначения.* Это гарантирует, что пакеты, которые получает хост, имеют подходящие IP-адреса источника/назначения.
- *Пакетам нужно делать обход подсетей.* Если пакету нужно покинуть подсеть, IP-адрес назначения проверяют внутренние маршрутизаторы AWS. В таком случае, если пакет содержит IP-адрес Pod'a, он не пройдет дальше.

В таких ситуациях можно воспользоваться протоколами туннелирования.

Инкапсуляция и туннелирование

Протоколы туннелирования дают возможность организовать Pod-сеть так, чтобы сети, на которой она основана, не были известны основные ее детали. Это достигается за счет инкапсуляции. Как можно догадаться по названию, инкапсуляция заключается в размещении одного (внутреннего) пакета внутри другого (внешнего). Во внутреннем пакете поля с IP-адресами источника и назначения указывают на приложение (Pod), а во внешнем — на IP-адреса хоста/узла. Когда пакет покидает узел, для основной сети он выглядит, как любые другие пакеты, так как данные, относящиеся к приложению, спрятаны внутри. Существует множество протоколов туннелирования, таких как VXLAN, Geneve и GRE. В Kubernetes одним из самых распространенных методов, которые используют сетевые подключаемые модули, стал VXLAN. На рис. 5.3 показано, как инкапсулированный пакет проходит по сети с помощью этого протокола.

Как видите, VXLAN помещает весь кадр Ethernet внутрь UDP-пакета. Вы фактически получаете полностью виртуальную сеть второго уровня, которую многие называют оверлейной. Сеть, которая находится уровнем ниже, ничего о ней не знает. Это одно из главных преимуществ протоколов туннелирования.

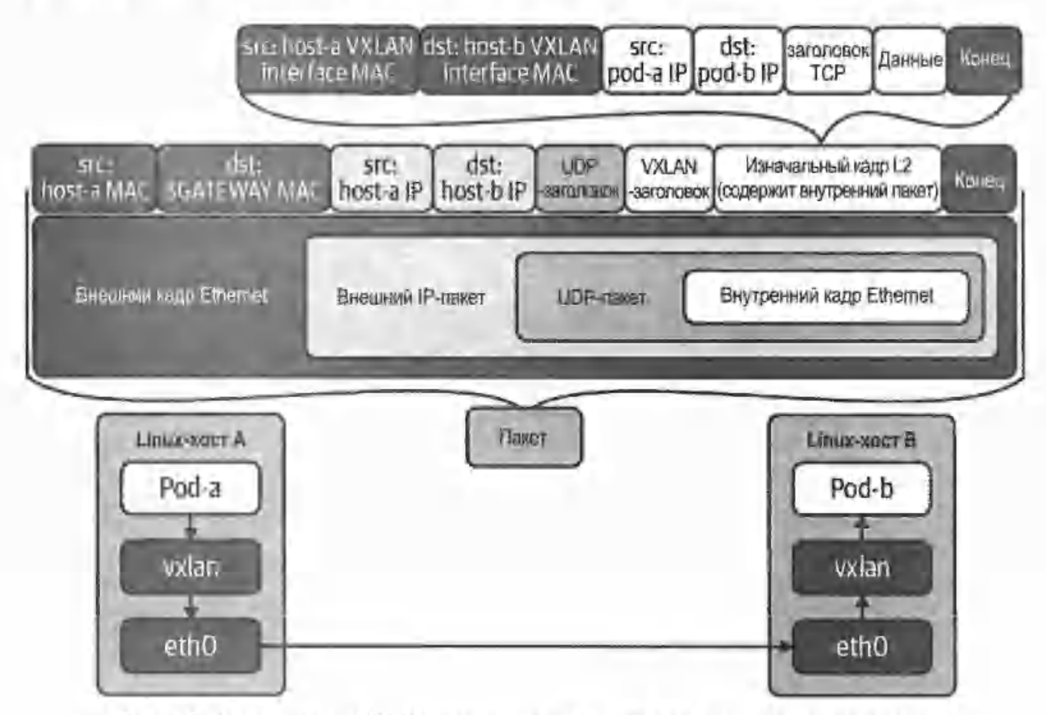


Рис. 5.3 Инкапсуляция VXLAN для передачи между хостами внутреннего пакета, предназначенного для рабочих заданий. Сеть интересует только внешний пакет, поэтому ей не нужно ничего знать об IP-адресах рабочих заданий и их маршрутах

Решение о том, использовать протокол туннелирования или нет, зачастую зависит от требований или возможностей вашего окружения. Преимущество инкапсуляции

в том, что она подходит для многих сценариев, так как оверлейная сеть скрывается от нижележащей. Однако у этого подхода есть несколько ключевых недостатков:

- *Может усложниться анализ и диагностика трафика.* Размещение одних пакетов внутри других может затруднить диагностику проблем в сети.
- *Упаковка/распаковка требует вычислительных ресурсов.* Перед тем как покинуть хост или попасть на хост, пакет должен быть инкапсулирован и, соответственно, декапсулирован. Это хоть и небольшие, но все же дополнительные накладные расходы по сравнению со стандартной маршрутизацией.
- *Пакеты будут более крупными.* Из-за упаковывания пакетов их размер увеличивается при передаче по сети. Это может потребовать изменения максимальной единицы передачи (англ. Maximum Transmission Unit или MTU; **https://oreil.ly/dzYBz**), чтобы сеть могла их уместить.

Маршрутизируемость приложений

В большинстве кластеров Pod-сети являются сугубо внутренними. Это означает, что Pod'ы могут взаимодействовать между собой напрямую, но внешние клиенты не могут обращаться к ним по их IP-адресам. Во многих случаях такое обращение не подходит, учитывая, что эти адреса временные. Вместо этого лучше положиться на механизмы обнаружения сервисов или балансировки нагрузки, которые скрывают внутренние IP-адреса. Огромное преимущество внутренних Pod-сетей — они *не* занимают драгоценное адресное пространство в вашей организации. Многие компании заботятся о том, чтобы внутренние адреса оставались уникальными. Поэтому, если вы попросите выделить подсеть `/16` (65 536 IP-адресов) для каждого кластера Kubernetes, который вы конфигурируете, на вас точно посмотрят с неприязнью!

Когда Pod'ы не доступны напрямую, трафик к ним можно направить несколькими способами. Обычно для этого в подсети из выделенных узлов развертывают контроллер Ingress. Он может перенаправлять пакеты, принимаемые его прокси-сервером, непосредственно к Pod'ам, так как он является участником Pod-сети. Некоторые облачные провайдеры даже предоставляют интеграцию с внешним балансировщиком нагрузки, которая автоматизирует этот процесс. Разнообразные модели маршрутизации входящего трафика и их недостатки будут описаны в *главе 6*.

Иногда требуется, чтобы Pod'ы были доступными из внешней, более крупной сети. Этого можно добиться двумя основными путями. Первый путь состоит в использовании сетевого подключаемого модуля, который напрямую интегрируется с нижележащей сетью. Например, VPC CNI от AWS (**https://github.com/aws/amazon-vpc-cni-k8s**) закрепляет несколько вторичных IP-адресов за каждым узлом и выделяет их для Pod'ов. В результате каждый Pod становится доступным снаружи, словно обычный сервер EC2. Главный недостаток этой модели — расход IP-адресов в вашей подсети или VPC. Второй вариант — доставлять маршруты к Pod'ам по протоколам маршрутизации вроде BGP, как было описано в разделе "Протоколы маршрутизации" данной главы. Некоторые подключаемые модули, использующие BGP, делают доступной снаружи лишь часть вашей Pod-сети, а не все пространство IP-адресов.



Pod-сеть следует делать доступной снаружи только в случае крайней необходимости. Как показывает наш опыт, это часто мотивировано наличием устаревших приложений. Представьте себе, к примеру, программный код на основе TCP, в котором клиент должен быть привязан к определенному внутреннему компоненту. Обычно мы рекомендуем обновить приложения в соответствии с парадигмой сетей контейнеров. Для этого нужно задействовать механизмы обнаружения сервисов и, возможно, изменить архитектуру внутреннего компонента так, чтобы тот не требовал привязки клиента к серверу (если это возможно). Предоставление доступа к Pod-сети может казаться простым решением, но за него приходится платить расходованием пространства IP-адресов, а также усложнением конфигурации IPAM и процесса распространения маршрутов.

IPv4 и IPv6

На сегодня подавляющее большинство кластеров используют сугубо IPv4. Но некоторые клиенты, такие как телекоммуникационные компании, в которых вопрос адресуемости приложений стоит остро, высказывают желание перейти на кластеры с сетями IPv6. Kubernetes 1.16 поддерживает IPv6 с помощью функции `dual-stack` (**https://oreil.ly/sj_jN**), которая на момент написания этих строк находится на стадии alpha-тестирования. Она позволяет конфигурировать в кластерах адресные пространства IPv4 и IPv6.

Если вам требуется протокол IPv6, то для его поддержки необходимо подготовить несколько компонентов:

- Для kube-apiserver и kubelet нужно включить функцию `feature-gate`, которая все еще находится на стадии alpha-тестирования.
- Нужно дополнительно сконфигурировать kube-apiserver, kube-controller-manager и kube-proxy, чтобы указать пространства IPv4 и IPv6.
- Вы должны использовать подключаемый модуль CNI с поддержкой IPv6 такой, как Calico (**https://projectca lico.org**) или Cilium (**https://cilium.io**).

Выполнив все перечисленные шаги, вы увидите, что для каждого объекта `Node` выделяется два диапазона IP-адресов:

```
spec:
  podCIDR: 10.30.0.0/24
  podCIDRs:
  - 10.30.0.0/24
  - 2002:1:1::/96
```

Механизм IPAM подключаемого модуля CNI самостоятельно определяет, какой диапазон был назначен каждой отдельной Pod-оболочке: IPv4, IPv6 или оба.

Шифрование трафика рабочих заданий

Трафик, которым обмениваются Pod'ы, по умолчанию шифруется редко (или вообще никогда). Это означает, что пакеты, отправленные по сети без шифрования, такого как TLS, могут быть прочитаны в виде обычного текста. Многие сетевые подключаемые модули поддерживают шифрование трафика. Например, Antrea

умеет шифровать данные с помощью IPsec (**https://oreil.ly/jqzCQ**) при использовании туннеля GRE. Calico может задействовать для этого копию WireGuard (**https://www.wireguard.com**), установленную на узле.

Применение шифрования может показаться очевидным решением. Но при этом нужно учитывать некоторые факторы. Мы советуем вам обратиться к вашим сетевым администраторам, чтобы понять, как происходит обмен трафиком между хостами на текущий момент. Шифруются ли данные, которые проходят от хоста к хосту внутри вашего ЦОД? Возможно, у вас уже функционируют какие-то другие механизмы шифрования? Например, взаимодействуют ли все ваши сервисы по TLS? Планируете ли вы использовать mesh-сети, в рамках которых прокси-серверы приложений применяют mTLS? Если да, то требуется ли дополнительное шифрование на уровне прокси-сервиса и CNI? Несомненно, шифрование повысит степень защиты, но в то же время оно усложнит администрирование и диагностику сети. И самое главное, шифрование и расшифровка пакетов повлияет на производительность и, следовательно, уменьшит вашу потенциальную пропускную способность.

Сетевая политика

Следующий логический шаг после конфигурации Pod-сети состоит в продумывании сетевой политики. Сетевая политика похожа на правила брандмауэра или группы безопасности, которые позволяют определить допустимый входящий и исходящий трафик. В состав основных сетевых API-интерфейсов Kubernetes входит NetworkPolicy API (**https://oreil.ly/1UV_3**). Политики можно добавлять в любой кластер, однако ответственность за их *реализацию* ложится на провайдера CNI. Это означает, что кластер, провайдер CNI которого не поддерживает NetworkPolicy (как, например, flannel; **https://github.com/coreos/flannel**), принимает объекты NetworkPolicy, но никак на них не реагирует. На сегодня большинство провайдеров CNI поддерживают NetworkPolicy в том или ином виде. А для тех, которые этого не делают, можно предусмотреть такие подключаемые модули, как Calico, способные работать в режиме, в котором они занимаются одним лишь обеспечением соблюдения политики.

Наличие NetworkPolicy внутри Kubernetes дает вам еще один уровень, на котором можно управлять правилами в стиле брандмауэра. Например, во многих сетях доступны правила для подсетей или хостов, реализованные в виде распределенного брандмауэра или механизма групп безопасности. Подобные решения хороши, но им ничего не известно о Pod-сетях. Это ограничивает степень гибкости, которую вам, возможно, хотелось бы иметь при создании правил для взаимодействия между приложениями на основе Pod'ов. Еще один важный аспект политики Kubernetes NetworkPolicy состоит в том, что она, как и большинство объектов Kubernetes, описывается декларативным образом, что, как нам кажется, намного проще по сравнению с большинством решений для управления брандмауэрами! В связи с этим мы, как правило, рекомендуем реализовывать сетевую политику на уровне Kubernetes, вместо того чтобы пытаться адаптировать существующие решения к этой новой парадигме. Это вовсе не означает, что вам нужно полностью отказаться от уже

имеющихся средств управления трафиком между хостами. А сетевую политику взаимодействия между приложениями целесообразно возложить на Kubernetes.

Решив использовать политики NetworkPolicy, вы должны понимать, что они действуют *в рамках пространства имен*. В отсутствие объектов NetworkPolicy весь входящий и исходящий трафик приложений в Kubernetes по умолчанию разрешен. При описании политики вы можете указать, к каким приложениям она относится. В этом случае поведение по умолчанию меняется на противоположное: любой исходящий и входящий трафик, не предусмотренный политикой, блокируется. Сказанное означает, что Kubernetes NetworkPolicy API определяет только допустимый трафик. Кроме того, в пространстве имен действует совокупность политик. Взгляните на объект `NetworkPolicy`, который определяет правила для входящего и исходящего трафика (листинг 5.2).

Листинг 5.2

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: team-netpol
  namespace: org-1
spec:
  podSelector: {} ❶
  policyTypes:
  - Ingress
  - Egress
  ingress: ❷
  - from:
    - ipBlock:
      cidr: 10.40.0.0/24
    ports:
    - protocol: TCP
      port: 80
  egress:
  - to: ❸
    ports:
    - protocol: UDP
      port: 53
  - to: ❹
    - namespaceSelector:
        matchLabels:
          name: org-2
    - podSelector:
        matchLabels:
          app: team-b
    ports:
    - protocol: TCP
      port: 80
```

❶ Пустое поле `podSelector` подразумевает, что эта политика действует для всех Pod'ов в данном пространстве имен. В качестве альтернативы вы можете выполнить сопоставление по метке.

❷ Это входящее правило разрешает принимать трафик из источников с IP-адресами в диапазоне 10.40.0.0/24, если он приходит по протоколу TCP на порт 80.

❸ Это исходящее правило разрешает приложениям отправлять DNS-трафик.

❹ Это исходящее правило разрешает отправлять только те пакеты, которые предназначены для приложений с меткой `team-b` в пространстве имен `org-2`. Кроме того, они должны передаваться по протоколу TCP на порт 80.

Согласно нашим наблюдениям, применение NetworkPolicy API постепенно сводится к определенным сценариям использования. Среди наиболее распространенных можно выделить такие:

- проверка сложных условий;
- получение IP-адресов из DNS-записей;
- правила L7 (хост, путь и т. д.);
- общекластерная политика, позволяющая определять глобальные правила вместо того, чтобы дублировать их в каждом пространстве имен.

Для удовлетворения этих потребностей подключаемые модули CNI предлагают собственные, более развитые API-интерфейсы управления политиками. Основной недостаток использования API-интерфейсов от конкретных провайдеров состоит в том, что ваши правила нельзя будет переносить между разными подключаемыми модулями. Мы исследуем примеры таких интерфейсов позже в этой главе, когда речь пойдет о Calico и Cilium.

Аспекты, связанные с сетью: итоги

В этом разделе мы обсудили ключевые аспекты организации сети, понимание которых позволит вам принять обоснованное решение о выборе стратегии сетевого взаимодействия ваших Pod'ов. Прежде чем приступать к подробному рассмотрению CNI и подключаемых модулей, еще раз перечислим некоторые ключевые вопросы:

- Насколько обширный диапазон IP-адресов в вашем кластере должен отводиться Pod'ам?
- Каким образом нижележащая сеть ограничивает возможности вашей будущей Pod-сети?
- Если вы используете управляемый сервис Kubernetes или готовое решение от поставщика, какие сетевые модули поддерживаются?
- Поддерживает ли ваша инфраструктура протоколы маршрутизации, такие как BGP?
- Могут ли по вашей сети проходить неинкапсулированные (стандартные) пакеты?

- Применение протоколов туннелирования (инкапсуляции) является желательным или необходимым?
- Нужно ли вам делать Pod'ы доступными снаружи?
- Требуется ли вашим приложениям поддержка IPv6?
- На каком уровне (или уровнях) вы планируете обеспечивать соблюдение сетевой политики или правил брандмауэра?
- Нужно ли вам шифровать трафик в вашей Pod-сети?

Сформулировав ответы на эти вопросы, вы можете с уверенностью приступать к изучению механизма, который позволяет подключить подходящую технологию для решения этих проблем, CNI (Container Networking Interface — интерфейс управления сетью контейнеров).

Интерфейс управления сетью контейнеров (CNI)

Все вопросы, которые мы обсуждали до сих пор, явно указывают на то, что разные сценарии использования требуют различных решений для управления сетью контейнеров. На ранних этапах развития Kubernetes в большинстве кластеров был сетевой подключаемый модуль под названием flannel (**https://github.com/coreos/flannel**). Со временем набрали популярность такие решения, как Calico (**https://www.projectcalico.org**) и прочие. Новые подключаемые модули начали предлагать разные подходы к созданию и администрированию сетей. Это в свою очередь способствовало разработке стандарта, описывающего то, как системы наподобие Kubernetes могли бы запрашивать сетевые ресурсы для своих рабочих заданий. Данный стандарт известен под названием CNI (Container Networking Interface — интерфейс управления сетью контейнеров; **https://github.com/containernet working/cni**). На сегодня этому интерфейсу соответствуют все параметры сети, совместимые с Kubernetes. По аналогии с CSI (Container Storage Interface — интерфейс хранилищ для контейнеров) и CRI (Container Runtime Interface — интерфейс среды выполнения контейнеров) это придает гибкость сетевому стеку нашей платформы приложений.

В спецификации CNI определено несколько ключевых операций:

- `ADD` — добавляет контейнер в сеть, возвращая соответствующие интерфейс(-ы), IP-адрес(-а) и пр.
- `DELETE` — удаляет контейнер из сети и освобождает все связанные с ним ресурсы.
- `CHECK` — проверяет корректность сетевой конфигурации и возвращает ошибку в случае обнаружения каких-либо проблем.
- `VERSION` — возвращает версию(-ии) CNI, которую(-ые) поддерживает подключаемый модуль.

Эта функциональность реализована в исполняемом файле, установленном на хосте. Агент kubelet обращается к подходящему модулю CNI в зависимости от конфигу-

рации, которую он ожидает от хоста. Пример такого конфигурационного файла приведен в листинге 5.3.

Листинг 5.3

```
{
  "cniVersion": "0.4.0", ❶
  "name": "dbnet", ❷
  "type": "bridge",
  "bridge": "cni0",
  "args": {
    "labels" : {
        "appVersion" : "1.0"
    }
  },
  "ipam": { ❸
    "type": "host-local",
    "subnet": "10.1.0.0/16",
    "gateway": "10.1.0.1"
  }
}
```

❶ Версия спецификации CNI, которую этот подключаемый модуль хочет использовать для взаимодействия.

❷ Драйвер CNI (исполняемый файл), которому нужно направлять запросы с конфигурацией сети.

❸ Драйвер IPAM, который нужно применить; указывается, если подключаемый модуль не имеет встроенной поддержки IPAM.



В директории conf драйвера CNI могут находиться разные конфигурационные файлы. Они проверяются в лексикографическом порядке, и используется только первый из них.

Помимо конфигурационных и исполняемых файлов CNI большинство подключаемых модулей размещают на каждом хосте Pod, который берет на себя обязанности, выходящие за рамки подключения интерфейса и IPAM. В их число входят подготовка маршрутов и программирование сетевых политик.

Установка CNI

Драйверы CNI должны быть установлены на каждом узле, принимающем участие в Pod-сети. Помимо этого, должна быть определена конфигурация CNI. Процесс установки обычно выполняется во время развертывания подключаемого модуля. Например, когда вы развертываете Cilium, создается объект `DaemonSet`, который размещает на каждом узле Pod `cilium`. Pod поддерживает команду `PostStart`, которая запускает встроенный скрипт `install-cni.sh`. Этот скрипт устанавливает два драй-

вера: сначала драйвер `loopback` для поддержки интерфейса `lo`, а затем драйвер `cilium`. На концептуальном уровне этот скрипт выполняется следующим образом (данный пример был существенно упрощен для краткости):

```
# Установка на хост драйверов CNI

# Установка драйвера loopback; может быть неудачной
cp /cni/loopback /opt/cin/bin/ || true

# Установка драйвера cilium
cp /opt/cni/bin/cilium-cni /opt/cni/bin/
```

После установки модуля агенту kubelet нужно как-то узнать, какой драйвер использовать. Для этого он заглянет внутрь директории /etc/cni/net.d/ (это можно настроить с помощью флага) в поиске конфигурации CNI. Тот же скрипт `install-cni.sh` добавляет в эту директорию файл, содержимое которого приведено в листинге 5.4.

Листинг 5.4

```
cat > /etc/cni/net.d/05-cilium.conf <<EOF
{
  "cniVersion": "0.3.1",
  "name": "cilium",
  "type": "cilium-cni",
  "enable-debug": ${ENABLE_DEBUG}
}
EOF
```

Чтобы продемонстрировать этот порядок операций, рассмотрим кластер из одного узла непосредственно после начальной конфигурации. Он был сконфигурирован с помощью `kubeadm`. Если вывести список всех запущенных Pod'ов, можно заметить, что среди них нет `core-dns` (листинг 5.5).

Листинг 5.5

```
NAMESPACE    NAME                            READY  STATUS   RESTARTS  AGE
kube-system  coredns-f9fd979d6-26lfr         0/1    Pending  0         3m14s
kube-system  coredns-f9fd979d6-zqzft         0/1    Pending  0         3m14s
kube-system  etcd-test                       1/1    Running  0         3m26s
kube-system  kube-apiserver-test             1/1    Running  0         3m26s
kube-system  kube-controller-manager-test    1/1    Running  0         3m26s
kube-system  kube-proxy-xhh2p                1/1    Running  0         3m14s
kube-system  kube-scheduler-test             1/1    Running  0         3m26s
```

Анализ журнальных записей kubelet на хосте (листинг 5.6), на котором запланирована работа `core-dns`, наглядно показывает, что среда выполнения контейнеров не запустила Pod по причине отсутствия конфигурации CNI.



Эта ситуация, в которой не запускается DNS, служит одним из самых распространенных признаков проблем с CNI, возникших после начальной конфигурации кластера. Еще одним симптомом являются узлы со статусом `NotReady`.

Листинг 5.6

```
# journalctl -f -u kubelet

-- Logs begin at Sun 2020-09-27 15:40:13 UTC. --
Sep 27 17:11:18 test kubelet[2972]: E0927 17:11:18.817089 2972 kubelet.go:2103]
Container runtime network not ready: NetworkReady=false
reason:NetworkPluginNotReady message:docker: network plugin is not ready: cni
config uninitialized
Sep 27 17:11:19 test kubelet[2972]: W0927 17:11:19.198643 2972 cni.go:239]
Unable to update cni config: no networks found in /etc/cni/net.d
```



Причина удачного запуска таких Pod'ов, как `kube-apiserver` и `kube-controller-manager` в том, что они используют нижележащую сеть, а не Pod-сеть, что ограждает их от проблем, которые наблюдаются у `core-dns`.

Для того чтобы развернуть в кластере Cilium, достаточно применить файл YAML из документации этого проекта. В результате на каждом узле будет развернут вышеупомянутый Pod `cilium` и запущен скрипт `cni-install.sh`. Если заглянуть в директории с исполняемым и конфигурационным файлами CNI, можно увидеть установленные компоненты:

```
# ls /opt/cni/bin/ | grep -i cilium
cilium-cni
# ls /etc/cni/net.d/ | grep -i cilium
05-cilium.conf
```

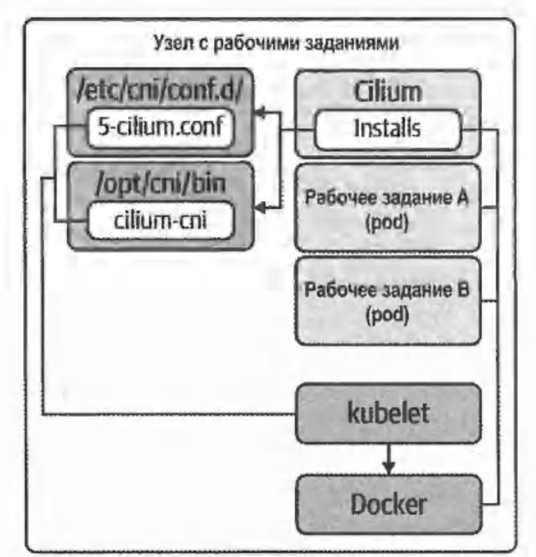


Рис. 5.4. Для выполнения контейнеров используется Docker. kubelet взаимодействует с CNI для присоединения сетевых интерфейсов и настройки сети Pod'ов

После установки среда выполнения контейнеров и kubelet работают так, как мы того ожидали. Что самое важное, работает Pod `core-dns`! На рис. 5.4 проиллюстрированы соотношения между компонентами, рассмотренными в этом разделе.

Здесь мы рассмотрели процесс установки на примере Cilium, но в большинстве подключаемых модулей применяется такая же модель развертывания. Основные факторы при выборе того или иного модуля были рассмотрены в разделе "Аспекты, связанные с сетью" данной главы. Имея это в виду, мы приступим к исследованию конкретных подключаемых модулей CNI, чтобы лучше понять разные подходы, которые в них реализованы.

Подключаемые модули CNI

Давайте рассмотрим несколько реализаций CNI. Если сравнивать с такими интерфейсами, как CRI, то окажется, что число различных вариантов у CNI одно из самых больших. В связи с этим мы не станем анализировать каждый из них. Можете самостоятельно изучить те, которые здесь не упоминаются. Наш выбор основан на том, насколько часто встречаются эти модули у наших клиентов, и достаточно ли они уникальны для того, чтобы продемонстрировать разнообразие подходов.



Pod-сеть — основополагающий элемент любого кластера Kubernetes. Поэтому вашему подключаемому модулю CNI будет отведена одна из ключевых ролей. Со временем у вас может возникнуть желание поменять свой модуль на другой. В таких случаях мы рекомендуем создавать кластеры заново, а не выполнять миграцию "по живому". Вы получите новый кластер с новым модулем CNI. Далее, в зависимости от вашей архитектуры и правил эксплуатации, вы сможете перенести в этот кластер свои приложения. Миграция в рамках существующего кластера тоже возможна, но она сопряжена с серьезным риском, и ее целесообразность следует тщательно сопоставить с предложенным нами вариантом.

Calico

Calico — это подключаемый модуль CNI, который хорошо себя зарекомендовал в облачно-ориентированной экосистеме. Project Calico (**https://www.projectcalico.org**) — это проект с открытым исходным кодом на основе данного модуля, а Tigera (**https://www.tigera.io**) — коммерческая компания, которая предоставляет возможности и поддержку уровня предприятия. Calico активно использует BGP для маршрутизации приложений по узлам и для обеспечения интеграции с более обширными сетями центра обработки данных. В дополнение к установке исполняемого файла CNI Calico запускает на каждом хосте агент `calico-node`. Последний содержит сервис BIRD для организации BGP-пиринга между узлами и агент Felix, который берет известные маршруты и заносит их в таблицы маршрутизации ядра. На рис. 5.5 показано, как это работает.

Для поддержки IPAM Calico по умолчанию соблюдает параметры `cluster-cidr`, описанные в разделе "Управление IP-адресами" данной главы. Но возможности этого модуля далеко не ограничиваются выделением диапазонов IP-адресов для каждого отдельного узла. Calico создает диапазоны, которые называют IP-пулами

(IPPool; **https://oreil.ly/-Nd-Q**). Это делает конфигурацию IPAM очень гибкой. В частности, поддерживаются такие возможности:

- выбор размера блока для каждого узла;
- выбор узлов, для которых действует IPPool;
- выделение IP-пулов для пространств имен, а не для отдельных узлов;
- настройка маршрутизации.

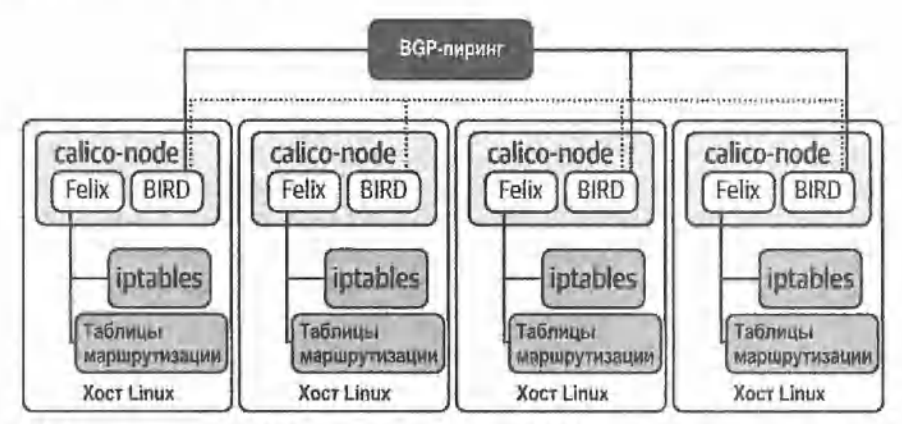


Рис. 5.5. Схема компонентов Calico, которые организуют BGP-пиринг для передачи маршрутов и, соответственно, программируют iptables и таблицы маршрутизации ядра

В сочетании с возможностью выделять сразу несколько пулов в каждом кластере это делает IPAM и сетевую архитектуру очень гибкими. По умолчанию кластеры используют один IP-пул, как показано в листинге 5.7.

Листинг 5.7

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: default-ipv4-ippool
spec:
  cidr: 10.30.0.0/16 ❶
  blockSize: 29 ❷
  ipipMode: Always ❸
  natOutgoing: true
```

❶ Диапазон IP-адресов Pod-сети кластера.

❷ Размер каждого диапазона уровня узла.

❸ Режим инкапсуляции.

Calico предлагает ряд методов маршрутизации пакетов внутри кластера, включая следующие:

- *Native* — отсутствие инкапсуляции пакетов.

- *IP-in-IP* — простая инкапсуляция. Один IP-пакет помещается внутрь другого.
- *VXLAN* — расширенная инкапсуляция. Внутри UDP-пакета инкапсулируется целый кадр L2, в результате чего создается виртуальная оверлейная сеть L2.

Выбор того или иного режима зачастую осуществляется с учетом того, что поддерживает ваша сеть. Как уже было описано в разделе "Протоколы маршрутизации", в большинстве случаев стандартные механизмы маршрутизации обеспечивают наилучшую производительность, наименьший размер пакета и самую простую диагностику. Однако во многих окружениях, особенно в тех, которые состоят из множества подсетей, такой режим невозможен. Методы инкапсуляции, особенно VXLAN, работают в большинстве окружений. Кроме того, режим VXLAN не требует использования BGP, что может быть выходом из ситуации там, где BGP-пиринг заблокирован. Одна из уникальных черт подхода к инкапсуляции, применяемого в Calico, состоит в том, что он действует исключительно для трафика, который проходит через границу подсети. По своей производительности этот подход сравним со стандартной маршрутизацией при использовании его внутри подсети, и в то же время он не нарушает маршрутизацию за ее пределами. Этого можно добиться, присвоив полю IP-пула `ipipMode` значение `CrossSubnet`. Данное поведение продемонстрировано на рис. 5.6.

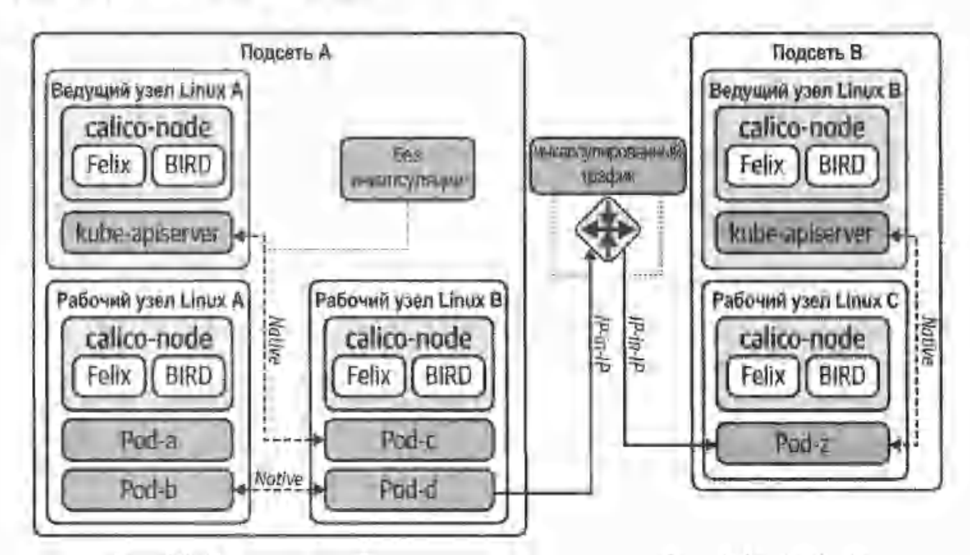


Рис. 5.6. Поведение трафика при включенном режиме CrossSubnet IP-in-IP

Развертывание Calico с включенным BGP по умолчанию не требует никакой дополнительной конфигурации благодаря встроенному BGP-сервису в Pod Calico. В более сложных архитектурах организации добавляют поддержку BGP в качестве отражателей маршрутов (route reflectors) (**https://tools.ietf.org/html/rfc4456**), которые иногда требуются в крупномасштабных системах, где не хватает стандартного подхода с полноценной mesh-сетью. Помимо отражателей маршрутов можно также настроить пиринг, чтобы взаимодействовать с сетевыми маршрутизаторами, что в свою очередь может сделать маршруты к Pod'ам известными на уровне всей сети в

целом. Все это настраивается с помощью определения пользовательского ресурса `BGPPeer` из состава Calico (листинг 5.8).

Листинг 5.8

```
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: external-router
spec:
  peerIP: 192.23.11.100 ❶
  asNumber: 64567 ❷
  nodeSelector: routing-option == 'external' ❸
```

❶ IP-адрес устройства, с которым выполняется (BGP)-пиринг.

❷ ID автономной системы кластера (**https://oreil.ly/HiXLN**).

❸ Определяет, какие узлы кластера должны иметь пиринг с этим устройством. Это необязательное поле. Если его не указать, конфигурация BGPPeer будет считаться *глобальной*. Глобальный пиринг не рекомендуется только в случае, если определенная группа узлов должна предоставлять уникальные возможности маршрутизации, такие как поддержка маршрутизируемых IP-адресов.

Если говорить о сетевой политике, то Calico предлагает полноценную реализацию Kubernetes NetworkPolicy API с двумя дополнительными пользовательскими ресурсами для расширения функциональности: (projectcalico.org/v3).NetworkPolicy (**https://oreil.ly/oMsCm**) и GlobalNetworkPolicy (**https://oreil.ly/3pUOs**). Эти API-интерфейсы из состава Calico похожи на Kubernetes NetworkPolicy, но поддерживают более развитые правила и богатые выражения для проверок. Кроме того, поддерживается упорядочивание политик и политика прикладного уровня (требует интеграции с Istio). Особенно полезен ресурс GlobalNetworkPolicy, так как он применяет политику на общекластерном уровне. Это упрощает реализацию таких моделей, как микросегментация, в которых весь трафик по умолчанию блокируется, а передача входящих/исходящих пакетов допускается в зависимости от потребностей приложения. Вы можете применить политику GlobalNetworkPolicy, которая блокирует весь трафик за исключением важнейших сервисов наподобие DNS. Затем на уровне пространства имен можно при необходимости открывать доступ к входящему и исходящему трафику. Без GlobalNetworkPolicy нам пришлось бы добавлять и администрировать запретительные правила в *каждом* пространстве имен.



Так сложилось, что Calico принимает решения о маршрутизации пакетов с учетом iptables. Для получения конечных точек Сервисов Calico полагается на код kube-proxy. Что касается сетевой политики, Calico программирует iptables, чтобы определить, может ли пакет попасть на хост или покинуть его. На момент написания этих строк в Calico появилась поддержка плоскости данных eBPF. Мы ожидаем, что со временем в нее будет перенесено больше возможностей, которые использует Calico.

Cilium

Cilium — это более новый подключаемый модуль CNI по сравнению с Calico. Среди всех альтернатив он первым стал использовать расширенный фильтр пакетов Berkley (англ. extended Berkeley Packet Filter или eBPF; **https://ebpf.io**). Это означает, что он может обрабатывать пакеты прямо внутри ядра, не переходя в пространство пользователя. Если сочетать его с eXpress Data Path (XDP; **https://oreil.ly/M3m6t**), в драйвере сетевого адаптера можно создавать хуки для принятия решений о маршрутизации непосредственно в момент получения пакета.

Технология eBPF применяется в крупных масштабах такими компаниями, как Facebook (**https://oreil.ly/agUXl**) и Netflix (**https://oreil.ly/Ubt1Q**). Благодаря этому проект Cilium может похвастаться расширенными возможностями в таких направлениях, как масштабируемость, наблюдаемость и безопасность. За счет этой глубокой интеграции с BPF такие распространенные задачи CNI, как обеспечение соблюдения политики NetworkPolicy больше не выполняются в пространстве пользователя. Вместо этого активное применение карт eBPF (**https://oreil.ly/4Rdvf**) позволяет быстро принимать решения и при этом масштабироваться с добавлением все новых правил. Общая схема системы с использованием Cilium показана на рис. 5.7.

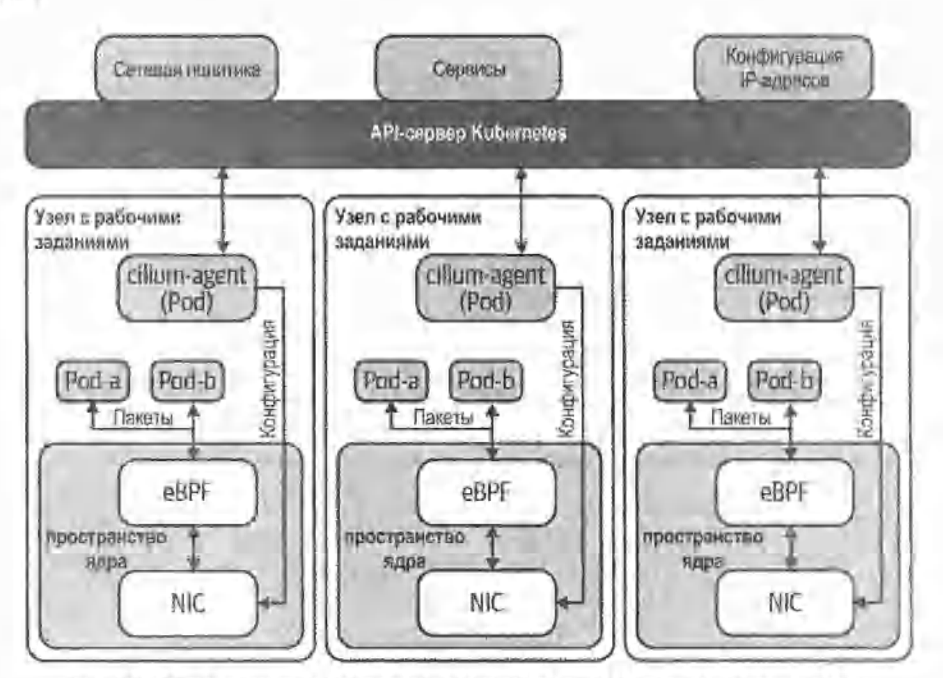


Рис. 5.7 Cilium взаимодействует с картами eBPF и программами на уровне ядра

Что касается IPAM, то эту обязанность Cilium может либо взять на себя, либо делегировать облачному провайдеру. В первом случае Cilium чаще всего выделяет диапазоны IP-адресов для каждого узла и по умолчанию управляет этими диапазонами независимо от процесса выделения узлов в Kubernetes. Адресация уровня узла дос-

тупна для настройки в пользовательском ресурсе `CiliumNode`. Этот подход дает повышенную гибкость при управлении IPAM и является предпочтительным. Если же вы желаете придерживаться стандартного процесса выделения диапазонов IP-адресов в Kubernetes, основанного на диапазонах Pod'ов, Cilium предлагает режим IPAM `kubernetes`. Он позволит учитывать диапазон адресов для Pod'ов, выделенный каждому узлу и доступный в объекте Node. Ниже показан пример объекта `CiliumNode`. Для каждого узла в кластере обычно отводится по одному такому объекту (листинг 5.9).

Листинг 5.9

```
apiVersion: cilium.io/v2
kind: CiliumNode
metadata:
  name: node-a
spec:
  addresses:
  - ip: 192.168.122.126 ❶
    type: InternalIP
  - ip: 10.0.0.245
    type: CiliumInternalIP
  health:
    ipv4: 10.0.0.78
  ipam:
    podCIDRs:
    - 10.0.0.0/24 ❷
```

❶ IP-адрес рабочего узла.

❷ Диапазон IP-адресов, выделенный этому узлу. Размер диапазона можно указать в конфигурационном файле Cilium с помощью параметра `cluster-pool-ipv4-mask-size: "24"`.

Подобно Calico, Cilium предлагает инкапсулированный и стандартный режимы маршрутизации. Первый задан по умолчанию. Cilium поддерживает протоколы туннелирования, такие как VXLAN и Geneve. Эта модель должна быть совместима с большинством сетей, в которых уже присутствует маршрутизация между хостами. Для работы в стандартном режиме сеть должна на каком-то уровне понимать маршруты Pod'ов. Например, Cilium может использовать для IPAM сервис ENI от AWS. В этом случае IP-адреса Pod-сетей становятся известными виртуальному частному облаку и маршрутизируемыми по своей сути. Чтобы включить стандартный режим и позволить Cilium управлять IPAM, в конфигурацию Cilium можно добавить параметр `auto-direct-node-routes: true` (при условии, что кластер размещен в том же сегменте L2). В результате Cilium соответствующим образом запрограммирует таблицы маршрутизации хоста. Если ваш кластер выходит за пределы одной сети L2, то для распределения маршрутов вам, возможно, придется воспользоваться дополнительными протоколами маршрутизации такими, как BGP.

Если говорить о сетевой политике, то Cilium может обеспечить соблюдение Kubernetes NetworkPolicy API (**https://oreil.ly/_WUKS**). В качестве альтернативы Cilium предлагает собственные ресурсы CiliumNetworkPolicy (**https://oreil.ly/EpkhJ**) и CiliumClusterwideNetworkPolicy (**https://oreil.ly/RtYH5**). Ключевое различие между ними состоит в охвате политики. CiliumNetworkPolicy действует на уровне пространства имен, а CiliumClusterwideNetworkPolicy — на уровне всего кластера. В обоих случаях мы получаем расширенные возможности по сравнению с Kubernetes NetworkPolicy. Политики, которые они поддерживают, могут быть основаны не только на метках (сетевой уровень L3), но и на запросах к DNS (прикладной уровень L7).

Большинство подключаемых модулей CNI не имеют никакого отношения к Сервисам, однако Cilium предлагает полноценную замену kube-proxy. Эта возможность встроена в агент `cilium-agent`, развертываемый на каждом узле. Для развертывания в этом режиме следует убедиться в том, что в вашем кластере нет kube-proxy, и что параметру Cilium `KubeProxyReplacement` присвоено значение `strict`. Cilium будет конфигурировать маршруты для Сервисов внутри карт eBPF, обеспечивая их получение со скоростью O(1). Для сравнения, kube-proxy реализует маршруты Сервисов в виде цепочек iptables, что может вызвать проблемы в широкомасштабных системах и/или при активном изменении состава сервисов. Кроме того, интерфейс командной оболочки Cilium предлагает удобные средства диагностики таких компонентов, как Сервисы и сетевые политики. Вместо того чтобы пытаться интерпретировать цепочки iptables, вы можете обращаться к системе следующим образом:

```
kubectl exec -it -n kube-system cilium-fmh8d -- cilium service list

ID   Frontend               Service Type   Backend
[...]
7 192.40.23.111:80          ClusterIP      1 => 10.30.0.28:80
                                           2 => 10.30.0.21:80
```

То, как модуль Cilium использует программы и карты eBPF, делает его чрезвычайно привлекательной и интересной реализацией CNI. Благодаря поддержке программ eBPF в Cilium интегрируется все больше функций, например, возможность извлекать сведения о перемещениях данных, нарушениях политики и др. Для получения и представления этой ценной информации был создан проект hubble (**https://github.com/cilium/hubble**). Он предоставляет пользовательский интерфейс и утилиту командной строки для операторов, используя поддержки программ eBPF в Cilium.

Напоследок нужно упомянуть о том, что возможности по работе с eBPF, доступные в Cilium, можно сочетать со многими существующими провайдерами CNI. Это достигается за счет выполнения Cilium в режиме цепочки CNI, в котором ответственность за маршрутизацию и IPAM перекладывается на другие подключаемые модули, такие как VPC CNI от AWS. В таком случае Cilium будет отвечать исключительно за функциональность, которая предоставляется разнообразными программами eBPF, включая наблюдаемость в сети, балансировку нагрузки и обеспечение соблюдения сетевой политики. Подобный подход может быть предпочтительным в

случаях, когда вы либо не можете использовать Cilium полноценно в своем окружении, либо желаете проверить возможности этого модуля параллельно с вашей текущей реализацией CNI.

AWS VPC CNI

VPC CNI от AWS демонстрирует совершенно другой подход по сравнению с тем, что мы видели до сих пор. Вместо того чтобы управлять Pod-сетью отдельно от сети узлов, этот подключаемый модуль полностью интегрирует Pod'ы в ту же сеть.

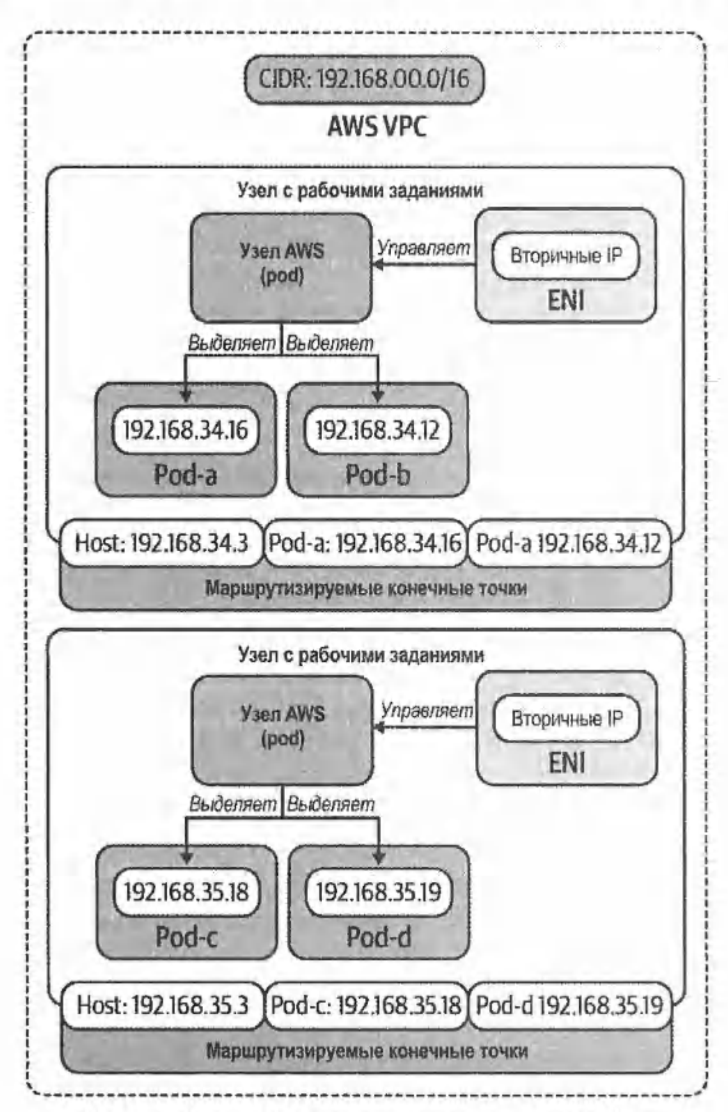


Рис. 5.8. Сервис IPAM отвечает за предоставление ENI и пула вторичных IP-адресов

Поскольку все происходит в рамках одной сети, нам больше не нужно беспокоиться о вопросах, связанных с распределением маршрутов или протоколами туннелирования. Когда Pod'у назначается IP-адрес, она становится частью сети по аналогии с тем, как это происходит с хостом EC2. Она подчиняется тем же таблицам маршрутизации (**https://oreil.ly/HYHHp**), что и любой хост в подсети. Amazon называет это стандартной сетью VPC (англ. native VPC networking).

Для поддержки IPAM сервис добавит в узел Kubernetes второй эластичный сетевой интерфейс (англ. elastic network interface или ENI; **https://oreil.ly/NBjs3**). Этот сервис станет хранить пул вторичных IP-адресов (**https://oreil.ly/vUGdI**), которые в конечном счете будут назначены Pod'ам. Число IP-адресов, доступных узлу, зависит от размера инстанса EC2. Эти адреса обычно являются "частными" внутри VPC. Как уже упоминалось ранее в этой главе, такой подход приводит к расходованию адресного пространства вашего VPC и делает системы IPAM более сложными по сравнению с полностью независимой Pod-сетью. Тем не менее, маршрутизация трафика и диагностика существенно упрощаются, так как мы не создаем новую сеть! На рис. 5.8 продемонстрирована конфигурация IPAM с использованием AWS VPC CNI.



Использование ENI повлияет на то, сколько Pod'ов вы сможете размещать на каждом узле. На странице AWS в GitHub есть список (**https://oreil.ly/jk_XL**), в котором указано соотношение типов инстансов и максимального числа Pod'ов.

Multus

До сих пор мы обсуждали конкретные подключаемые модули CNI, которые добавляют интерфейс в Pod'ы, делая их тем самым доступными в сети. Но что, если Pod должен быть подключен сразу к нескольким сетям? Для этого существует подключаемый модуль CNI Multus. Хоть и нечасто, но в сфере телекоммуникаций встречаются случаи, когда для маршрутизации трафика в определенную выделенную сеть необходимо предусмотреть виртуализацию сетевых функций (англ. Network Function Virtualization или NFV).

Multus можно считать реализацией CNI, которая делает возможным одновременное использование нескольких других реализаций. В этой модели Multus играет роль подключаемого модуля CNI, с которым взаимодействует Kubernetes. Его конфигурация предполагает наличие сети по умолчанию, на которую обычно ложится организация взаимодействия между Pod'ами. Это даже может быть один из подключаемых модулей, которые мы обсуждали в данной главе! Multus также позволяет конфигурировать вторичные сети путем задания дополнительных подключаемых модулей, с помощью которых можно добавлять в Pod еще один интерфейс. Pod'ы могут иметь аннотации наподобие `k8s.v1.cni.cncf.io/networks: sriov-conf` для присоединения дополнительной сети. Эта конфигурация показана на рис. 5.9.

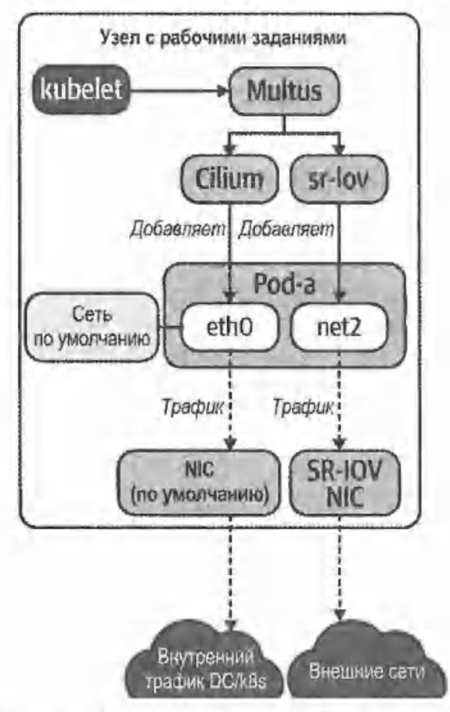


Рис. 5.9. Движение трафика в конфигурации Multus с несколькими сетями

Дополнительные подключаемые модули

Существует обширная система подключаемых модулей, и мы обсудили лишь небольшую ее часть. Тем не менее, те из них, которые были рассмотрены в этой главе, иллюстрируют некоторые ключевые отличия. Большинство альтернатив применяют другие подходы к организации сети, однако многие основные принципы остаются прежними. В приведенном далее списке перечислены некоторые дополнительные подключаемые модули с кратким описанием того, как в них организована сеть:

- *Antrea* ***(https://antrea.io/docs)*** — плоскость данных основана на Open vSwitch (**https://www.openvswitch.org**). Предлагает высокопроизводительную маршрутизацию и возможность анализировать данные, проходящие по сети.
- *Weave* ***(https://www.weave.works/oss/net)*** — оверлейная сеть, предоставляющая множество механизмов для маршрутизации трафика, например, быстрые пути к данным с использованием модулей OVS, которые позволяют не выносить обработку пакетов за пределы ядра.
- *flannel* ***(https://github.com/coreos/flannel)*** — простая сеть L3 (сетевой уровень) для Pod'ов. Один из старейших подключаемых модулей CNI. Поддерживает много внутренних компонентов, хотя чаще всего применяется в сочетании с VXLAN.

Резюме

Сетевая экосистема Kubernetes контейнеров предоставляет широкий выбор вариантов. И это хорошо! Как мы продемонстрировали в этой главе, требования к сети в разных организациях могут существенно различаться. Выбор подключаемого модуля CNI, вероятно, будет одним из самых основополагающих решений для вашей будущей платформы приложений. При исследовании многочисленных вариантов легко растеряться, но мы настоятельно советуем вам как следует разобраться в сетевых требованиях ваших приложений и окружения. Если глубоко вникнуть в суть этого аспекта, выбор подходящего сетевого модуля не должен вызвать никаких затруднений!

ГЛАВА 6

Маршрутизация сервисов

Маршрутизация сервисов — это одна из важнейших возможностей платформы, основанной на Kubernetes. Слой сети контейнеров берет на себя низкоуровневые механизмы, соединяющие Pod'ы, однако разработчикам нужны высокоуровневые средства связывания сервисов между собой (т. е. маршрутизация сервисов внутри системы). Маршрутизация сервисов включает в себя три компонента, которые предоставляют такие механизмы: Сервисы, объекты Ingress и mesh-сеть.

Сервисы дают возможность обращаться с группой Pod'ов как с единым целым (т. е. как с сетевым сервисом). Они предоставляют балансировку нагрузки и средства маршрутизации, что позволяет горизонтально масштабировать приложения по всему кластеру. Более того, они включают в себя механизмы обнаружения сервисов, с помощью которых приложения могут находить свои зависимости и взаимодействовать с ними. Наконец, Сервисы также обеспечивают механизмы сетевого и транспортного уровней (L3 и L4) для предоставления сетевым клиентам, находящимся за пределами кластера, доступа к приложениям.

Объект Ingress отвечает за маршрутизацию внешнего трафика в кластере. Он служит точкой доступа к приложениям, выполняющимся в кластере, в основном HTTP- и HTTPS-сервисам. Он предоставляет возможности балансировки прикладного уровня (L7), позволяя маршрутизировать трафик более гибко в сравнении с Сервисами. Балансировка трафика осуществляется контроллером Ingress, который должен быть установлен в кластере. Эти контроллеры используют такие прокси-серверы, как Envoy, NGINX или HAProxy. Они получают конфигурацию Ingress с помощью API-интерфейса Kubernetes и настраивают прокси-сервер соответствующим образом.

Mesh-сеть — это прослойка, предоставляющая такие возможности, как маршрутизация, безопасность и наблюдаемость сервисов. Она в основном занимается маршрутизацией внутри системы, но некоторые реализации могут брать на себя обработку внешнего трафика. Сервисы в mesh-сети взаимодействуют между собой через прокси-серверы, которые дополняют сетевые соединения. Применение прокси-серверов делают mesh-сети мощным решением, позволяя им дополнять функциональность приложений без изменения исходного кода.

Эта глава посвящена средствам маршрутизации сервисов, которые крайне важны в платформах Kubernetes в условиях эксплуатации. Вначале мы обсудим Сервисы, их различные типы и то, как они реализованы. Затем мы исследуем объекты Ingress, контроллеры Ingress и разные факторы, которые необходимо учитывать при использовании их в реальных условиях. В конце главы речь пойдет о mesh-сетях, о том, как они устроены в Kubernetes, и на что следует обращать внимание при их внедрении в платформу.

Сервисы Kubernetes

Ресурс Service из состава Kubernetes является основополагающим средством маршрутизации сервисов. Сервис — это сетевая абстракция, обеспечивающая балансировку нагрузки между несколькими Pod'ами. В большинстве случаев приложения, выполняемые в кластере, взаимодействуют между собой посредством объектов Service. Это предпочтительная альтернатива использованию IP-адресов Pod'ов ввиду взаимозаменяемости последних.

В этом разделе мы проведем краткий обзор Сервисов Kubernetes и различных их типов. Мы также рассмотрим объекты Endpoints — еще один ресурс Kubernetes, тесно связанный с сервисами. Затем речь пойдет об аспектах реализации Сервисов и о kube-proxy. В заключение мы обсудим механизмы обнаружения Сервисов и характеристики внутрикластерного DNS-сервера, на которые нужно обращать внимание.

Компонент Service

Ресурс Service — это один из основных API-интерфейсов Kubernetes, который балансирует трафик между разными Pod'ами на сетевом и транспортном уровнях модели OSI (L3 и L4). Сервис берет пакет с IP-адресом и портом назначения и направляет его к внутреннему Pod'у.

У балансировщиков нагрузки, как правило, есть внешняя часть и внутренний пул адресов. То же самое можно сказать и о Сервисах. Снаружи Сервис использует ClusterIP — виртуальный IP-адрес (англ. Virtual IP address или VIP), доступный изза пределов кластера. С помощью VIP приложения взаимодействуют с Сервисом. Внутренний пул представляет собой набор Pod'ов, которые соответствуют Pod-селектору Сервиса. Эти Pod'ы принимают трафик, направленный к ClusterIP. На рис. 6.1 изображена внешняя часть Сервиса и его внутренний пул.

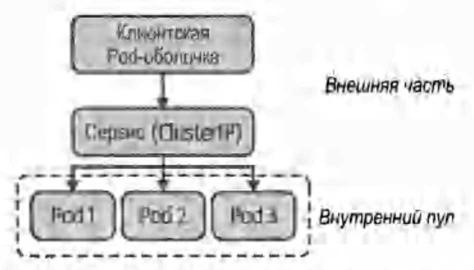


Рис. 6.1. У Сервиса есть внешняя часть и внутренний пул. Роль внешней части играет ClusterIP, а внутри находится набор Pod'ов

Управление IP-адресами Сервисов

Как уже обсуждалось в предыдущей главе, при развертывании Kubernetes нужно сконфигурировать два диапазона IP-адресов: один для Pod'ов в кластере, а другой — для Сервисов. С помощью последнего Kubernetes присваивает Сервисам значения ClusterIP.

API-сервер занимается управлением IP-адресами (IPAM) Сервисов Kubernetes. Когда вы создаете Сервис, API-сервер (с помощью etcd) выделяет IP-адреса из соответствующего диапазона и записывает их в поле ClusterIP объекта Service.

ClusterIP можно также указать в спецификации Сервиса при его создании. В этом случае API-сервер позаботится о том, чтобы запрошенные IP-адреса были доступными и находились в заданном диапазоне Сервиса. С другой стороны, явное назначение ClusterIP является порочной практикой.

Ресурс Service

Ресурс Service содержит конфигурацию заданного Сервиса, включая его название, тип, порты и т. д. В листинге 6.1 показано определение Сервиса в формате YAML под названием `nginx`.

Листинг 6.1. Определение Сервиса, открывающее доступ к NGINX по адресу ClusterIP

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector: ❶
    app: nginx
  ports: ❷
    - protocol: TCP ❸
      port: 80 ❹
      targetPort: 8080 ❺
clusterIP: 172.21.219.227 ❻
type: ClusterIP
```

❶ Pod-селектор, с помощью которого Kubernetes ищет Pod'ы, которые принадлежат данному Сервису.

❷ Порты, доступные посредством этого Сервиса.

❸ Сервисы Kubernetes поддерживают протоколы TCP, UDP и SCTP.

❹ Порт, по которому можно обратиться к Сервису.

❺ Порт, который прослушивает внутренний Pod; может отличаться от порта, который предоставляется Сервисом (поле `port` в данном примере).

❻ Адрес ClusterIP, который Kubernetes выделяет этому Сервису.

Pod-селектор Сервиса определяет Pod'ы, которые этому Сервису принадлежат. Pod-селектор представляет собой набор пар вида "ключ – значение", которые Kubernetes сопоставляет с Pod'ами, размещенными в том же пространстве имен, что и Сервис. Если метки Pod'а содержат те же пары "ключ – значение", Kubernetes добавляет ее IP-адрес во внутренний пул Сервиса. Управлением внутренним пулом занимается контроллер Endpoints с использованием одноименных ресурсов. Далее в этой главе мы рассмотрим объекты Endpoints более подробно.

Типы Сервисов

До сих пор мы в основном обсуждали Сервисы типа ClusterIP, который назначается по умолчанию. Kubernetes поддерживает другие типы Сервисов, обладающие дополнительными возможностями. В этом разделе мы их опишем и покажем, чем они могут быть полезны.

ClusterIP

Мы уже упоминали Сервисы этого типа в предыдущих разделах. Сервис ClusterIP создает виртуальный IP-адрес (англ. Virtual IP address или VIP), привязанный к одной или нескольким внутренним Pod'ам. Обычно VIP доступен только приложениям, выполняемым внутри кластера. Сервис ClusterIP показан на рис. 6.2.

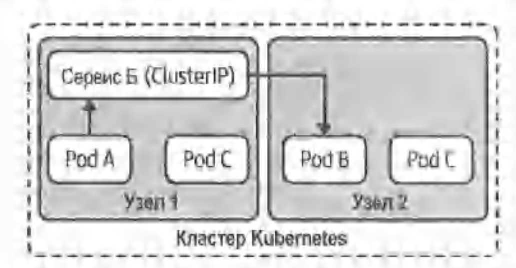


Рис. 6.2. Сервис ClusterIP — это VIP, доступный приложениям, выполняемым внутри кластера

NodePort

Тип NodePort подходит в ситуациях, когда вам нужно сделать Сервис доступным для клиентов, которые находятся вне кластера, например, существующим приложениям, работающим внутри ВМ, или пользователям веб-приложений.

Как можно догадаться по названию, Сервис NodePort доступен на определенном порту во всех узлах кластера. Этот порт выбирается случайным образом из диапазона, который можно сконфигурировать. После его назначения все узлы в кластере начинают прослушивать направленные к нему соединения. Сервис типа NodePort показан на рис. 6.3.

Основная трудность при использовании типа NodePort состоит в том, что для обращения к Сервису клиент должен знать номер его порта и IP-адрес как минимум одного узла в кластере. Это проблематично, поскольку узел может выйти из строя или перестать быть частью кластера.

Для решения этой проблемы перед Сервисом NodePort обычно размещают внешний балансировщик нагрузки, который освобождает клиента от необходимости знать IP-адреса узлов кластера или номер порта Сервиса. Балансировщик нагрузки играет роль единой точки входа в Сервис.

Недостаток этого решения в том, что вам придется обслуживать внешние балансировщики нагрузки и постоянно обновлять их конфигурацию. Разработчик развернул новый Сервис NodePort? Создайте новый балансировщик нагрузки. В кластере появился новый узел? Добавьте его во внутренний пул всех балансировщиков.

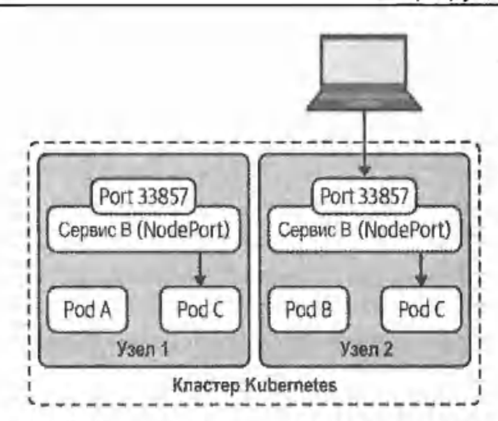


Рис. 6.3 Сервис NodePort открывает случайный порт на всех узлах кластера. Внешние клиенты могут обращаться к Сервису через этот порт

В большинстве случаев у NodePort есть более подходящие альтернативы. Одна из них — Сервис LoadBalancer, который мы обсудим далее. Еще одним вариантом могут быть контроллеры Ingress, которые будут рассмотрены позже в этой главе в разделе "Ingress".

LoadBalancer

Сервис типа LoadBalancer является попыткой решить некоторые из проблем NodePort. Внутри он основан на NodePort, но имеет дополнительную функциональность, реализованную с помощью контроллера.

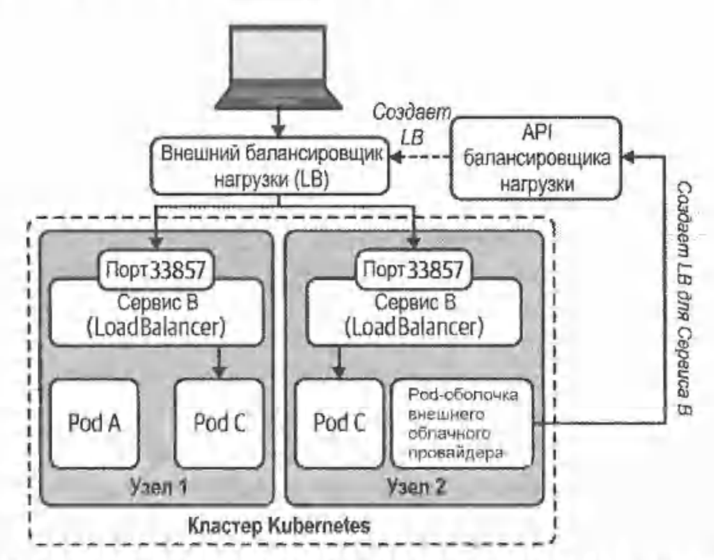


Рис. 6.4. Сервис LoadBalancer использует средства интеграции с облачным провайдером для создания внешнего балансировщика нагрузки, который направляет трафик к портам узлов. На самих узлах данный Сервис ничем не отличается от NodePort

Контроллер (или средство интеграции с облачным провайдером) отвечает за автоматическое связывание Сервиса NodePort с внешним балансировщиком нагрузки.

Иными словами, контроллер выполняет создание, администрирование и конфигурацию внешних балансировщиков нагрузки в ответ на изменение параметров Сервисов LoadBalancer в кластере. Для этого он обращается к API-интерфейсу, который выделяет или конфигурирует балансировщики. Это взаимодействие изображено на рис. 6.4.

У Kubernetes есть встроенные контроллеры для нескольких облачных провайдеров, включая Amazon Web Services (AWS), Google Cloud и Microsoft Azure. Эти интегрированные контроллеры обычно называют встроенными облачными провайдерами, так как они реализованы в рамках дерева исходного кода Kubernetes.

По мере развития проекта Kubernetes начали возникать альтернативные, внешние облачные провайдеры. Это позволило поставщикам балансировщиков нагрузки предоставлять собственные реализации управляющего цикла для Сервиса LoadBalancer. На сегодня Kubernetes имеет поддержку как встроенных, так и внешних провайдеров. Однако сообщество быстро переходит на последние, так как встроенные провайдеры объявлены устаревшими.

Сервисы LoadBalancer без облачного провайдера

Если использовать Kubernetes без интеграции с облачным провайдером, можно заметить, что Сервис LoadBalancer остается в состоянии ожидания ("Pending"). Отличный пример этого — развертывание платформы на физическом оборудовании. В таком случае у вас может быть возможность воспользоваться MetalLB для поддержки Сервисов LoadBalancer.

MetalLB (**https://metallb.universe.tf**) — это проект с открытым исходным кодом, который обеспечивает поддержку Сервисов типа LoadBalancer на физическом оборудовании. MetalLB размещается в кластере и может работать в одном из двух режимов. В канальном режиме (L2) один из узлов кластера становится ведущим и начинает отвечать на ARP-запросы к внешним IP-адресам Сервисов LoadBalancer. Как только трафик доходит до ведущего узла, его обработкой занимается kube-proxy. Если один ведущий узел выходит из строя, его роль берет на себя другой и начинает обслуживать запросы. Большой недостаток этой модели состоит в том, что она не предоставляет никаких реальных средств балансировки нагрузки, учитывая, что на все ARP-запросы отвечает один узел.

Второй режим работы предусматривает использование BGP для пиринга с сетевыми маршрутизаторами. Посредством пиринга MetalLB объявляет о внешних IP-адресах Сервисов LoadBalancer. Как и в канальном режиме, за маршрутизацию трафика от одного из узлов кластера к внутреннему Pod'у отвечает kube-proxy. Режим BGP является ответом на недостатки режима L2 и позволяет балансировать трафик между несколькими узлами, вместо того чтобы ограничиваться одним ведущим.

Если вам нужна поддержка Сервисов LoadBalancer, то MetalLB может стать хорошим решением. Но в большинстве случаев вы можете обойтись и без нее. Например, если большую долю ваших приложений составляют HTTP-сервисы, для балансировки и доставки трафика к ним можно воспользоваться контроллером Ingress.

ExternalName

Сервисы типа ExternalName не занимаются балансированием нагрузки или проксированием. Их основная обязанность состоит в предоставлении механизма обнаружения сервисов, реализованного в DNS-сервере кластера. ExternalName привязывает Сервис кластера к доменному имени. Ввиду отсутствия балансировки нагрузки у Сервисов этого типа нет ClusterIP.

Сервисы ExternalName можно применять для разных целей. Например, они хорошо подходят для поэтапной миграции приложений. Если вы переносите компоненты своего приложения в Kubernetes, оставляя при этом некоторые его зависимости снаружи, то Сервис ExternalName может служить для них мостом на время миграции. Завершив перенос всего приложения, вы можете поменять тип Сервиса на ClusterIP, не внося никаких изменений в уже имеющееся развертывание.

Несмотря на пользу, которую может принести ExternalName в нестандартных ситуациях, это, наверное, наименее распространенный тип Сервисов.

Неуправляемые Сервисы

Сервисы этого типа, как и ExternalName, не выделяют ClusterIP и не обеспечивают балансировку нагрузки. Они служат лишь средством регистрации других Сервисов и их конечных точек в API-интерфейсе Kubernetes и DNS-сервере кластера.

Неуправляемые Сервисы подходят в ситуациях, когда приложениям нужно подключаться к определенным репликам или Pod'ам сервиса. Такие приложения могут использовать механизм обнаружения сервисов для поиска IP-адресов всех Pod'ов заданного объекта Service с последующим подключением к нужным им Pod'ам.

Поддерживаемые протоколы взаимодействия

Сервисы Kubernetes поддерживают определенный набор протоколов: TCP, UDP и SCTP. Каждый порт, указанный в ресурсе `Service`, состоит из номера и протокола. Сервисы могут открывать доступ сразу к нескольким портам, и у каждого из них может быть свой протокол. Например, в листинге 6.2 приведен фрагмент кода в формате YAML с определением Сервиса `kube-dns`. Обратите внимание на то, что в поле `ports` содержатся TCP-порт 53 и UDP-порт 53.

Листинг 6.2

```
apiVersion: v1
kind: Service
metadata:
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: KubeDNS
  name: kube-dns
  namespace: kube-system
```

```
spec:
  clusterIP: 10.96.0.10
  ports:
  - name: dns
    port: 53
    protocol: UDP
    targetPort: 53
  - name: dns-tcp
    port: 53
    protocol: TCP
    targetPort: 53
  - name: metrics
    port: 9153
    protocol: TCP
    targetPort: 9153
  selector:
    k8s-app: kube-dns
  type: ClusterIP
```

Протоколы и диагностика Сервисов

Для диагностики Сервисов при работе с Kubernetes вы могли использовать команду `ping`. И вы, наверное, обнаружили, что все пакеты, которые вы отправляли Сервисам с ее помощью, терялись. Дело в том, что эта команда использует датаграммы ICMP, которые не поддерживаются Сервисами Kubernetes.

Когда речь идет о диагностике Сервисов, вместо `ping` необходимо применять альтернативные средства. Если вам нужно проверить сетевое соединение, выбирайте такие инструменты, которые работают с протоколами Сервиса. Например, для диагностики веб-сервера можно воспользоваться утилитой `telnet`, которая позволяет проверить, можно ли соединиться с этим сервером по TCP.

Еще один метод быстрой диагностики состоит в подтверждении того, что Pod-селектор охватывает как минимум один Pod, для этого можно проверить соответствующий ресурс Endpoints. Некорректные селекторы — распространенная проблема при работе с Сервисами.

Как уже упоминалось ранее, Сервисы балансируют трафик между Pod'ами. Ресурс Service API представляет клиентскую часть балансировщика нагрузки. Внутренняя часть (набор Pod'ов, которые скрываются за балансировщиком) отслеживается ресурсом и контроллером Endpoints, которые мы обсудим далее.

Endpoints

Ресурс Endpoints (конечные точки) — это еще один объект API-интерфейса, который участвует в реализации Сервисов Kubernetes. У каждого ресурса Service есть родственный ресурс Endpoints. Если вернуться к аналогии с балансировщиком нагрузки, то объект Endpoints можно считать пулом IP-адресов, которые принимают трафик. Соотношение между Service и Endpoints показано на рис. 6.5.

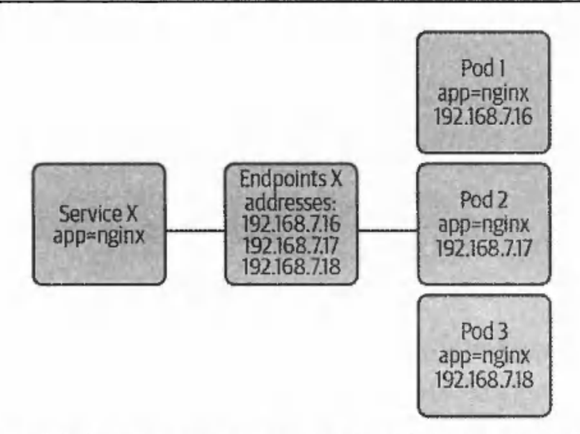


Рис. 6.5. Соотношение между ресурсами Service и Endpoints

Ресурс *Endpoints*

Листинг 6.3 иллюстрирует ресурс `Endpoints` для Сервиса `nginx` (некоторые лишние поля были опущены):

Листинг 6.3

```
apiVersion: v1
kind: Endpoints
metadata:
  labels:
    run: nginx
  name: nginx
  namespace: default
subsets:
- addresses:
- ip: 10.244.0.10
  nodeName: kube03
  targetRef:
    kind: Pod
    name: nginx-76df748b9-gblnn
    namespace: default
- ip: 10.244.0.9
  nodeName: kube04
  targetRef:
    kind: Pod
    name: nginx-76df748b9-gb7wl
    namespace: default
ports:
- port: 8080
  protocol: TCP
```

В этом примере Сервис `nginx` основан на двух Pod'ах, между которыми распределяется сетевой трафик, направляемый к ClusterIP `nginx`. Также обратите внимание, что здесь назначен порт 8080 вместо 80 (он соответствует полю `targetPort`, указанному в определении Сервиса). Это порт, который прослушивают внутренние Pod'ы.

Контроллер *Endpoints*

Интересной особенностью ресурса `Endpoints` является то, что Kubernetes создает его автоматически, когда вы создаете свой Сервис. Это несколько отличается от поведения других API-ресурсов, с которыми вы обычно имеете дело.

Контроллер Endpoints отвечает за создание и обслуживание одноименных объектов. Каждый раз, когда вы создаете Сервис, этот контроллер формирует родственный ресурс Endpoints. Что еще важнее, он также при необходимости обновляет список IP-адресов внутри объекта `Endpoints`.

Для поиска Pod'ов, которые принадлежат Сервису, данный контроллер использует Pod-селектор этого Сервиса. Получив набор Pod'ов, контроллер берет их IP-адреса и вносит соответствующие изменения в ресурс Endpoints.

Адреса в ресурсе Endpoints делятся на две группы: готовые (`addresses`) и неготовые (`notReadyAddresses`). Контроллер Endpoints определяет, какие из них готовы, путем проверки условия Ready соответствующего Pod'a. Условие Ready в свою очередь зависит от нескольких факторов. Один из них, к примеру, состоит в том, была ли запланирована работа Pod'a. Если Pod находится в состоянии ожидания (не запланирован), условие Ready не выполняется (false). В конечном счете Pod считается готовым, когда он запущен и проходит проверку готовности.

Готовность Pod'ов и проверки готовности

В предыдущем разделе мы обсуждали, как контроллер Endpoints определяет, готов ли IP-адрес Pod'a принимать трафик. Но откуда Kubernetes знает, готов Pod или нет?

Существует два взаимодополняющих метода, с помощью которых Kubernetes определяет готовность Pod'ов:

- *Информация о платформе* — система Kubernetes владеет информацией о приложениях, которыми она управляет. Например, она знает, было ли выполнение Pod'a на узле успешно запланировано. Ей также известно, работают ли контейнеры этого Pod'a.
- *Проверки работоспособности* — разработчики могут сконфигурировать проверки готовности для своих приложений. В этом случае kubelet периодически проверяет приложение, чтобы определить, готово ли оно к приему трафика. Проверка Pod'ов на готовность — более действенный подход по сравнению с использованием информации о платформе, так как мы можем проверять факторы, относящиеся к конкретным приложениям. Например, мы можем узнать, завершило ли приложение свой внутренний процесс инициализации.

Проверки готовности являются необходимыми. Без них кластер мог бы направлять трафик к приложениям, которые неспособны с ним справиться, что приводило бы к возникновению ошибок в приложениях и недовольству конечных пользователей. Позаботьтесь о том, чтобы для всех приложений, которые вы развертываете в Kubernetes, были определены проверки готовности. В *главе 14* мы продолжим обсуждение этой темы.

Ресурс *EndpointSlices*

Ресурс `EndpointSlices` — это механизм оптимизации, реализованный в Kubernetes версии 1.16. Он помогает решить проблемы с масштабированием, которые могут возникать у ресурса Endpoints при работе в крупных кластерах (**https://oreil.ly/H8rHC**). Давайте исследуем эти проблемы и посмотрим, как с ними борется `EndpointSlices`.

Чтобы реализовать Сервисы и сделать их доступными в сети, каждый узел в кластере следит за Endpoints API и подписывается на изменения. Чтобы вступить в силу, обновления, применяемые к ресурсу Endpoints, должны распространяться по всем узлам. Хорошим примером является событие масштабирования. Каждый раз, когда в ресурсе Endpoints меняется набор Pod'ов, API-сервер рассылает весь обновленный объект всем узлам кластера.

Этот способ применения Endpoints API плохо масштабируется в крупных кластерах по нескольким причинам:

- Большие кластеры содержат много узлов. Чем больше узлов в кластере, тем больше обновлений нужно рассылать при изменении объектов `Endpoints`.
- Чем больше кластер, тем больше Pod'ов (и Сервисов) в нем можно разместить. И по мере увеличения числа Pod'ов растет и частота обновлений ресурса `Endpoints`.
- Размер ресурса Endpoints растет вместе с количеством Pod'ов, принадлежащих Сервису. Чем больше объекты Endpoints, тем больше ресурсов сети и хранилища им нужно.

Средство `EndpointSlices` решает эти проблемы, разделяя набор конечных точек на несколько ресурсов. Вместо того чтобы хранить IP-адреса всех Pod'ов в одном ресурсе Endpoints, Kubernetes распределяет их между разными объектами `EndpointSlice`. По умолчанию объекты `EndpointSlice` могут содержать не больше 100 конечных точек.

Чтобы лучше понять результат использования `EndpointSlice`, рассмотрим конкретный пример. Представьте себе Сервис с 10 000 конечных точек, который потребовал бы 100 объектов `EndpointSlice`. Если одна из этих конечных точек удаляется (например, в процессе уменьшения масштаба кластера), то API-сервер рассылает каждому узлу затронутый этим фактом объект `EndpointSlice` (который содержит 100 конечных точек). Это намного эффективней, чем рассылать один ресурс Endpoints с тысячами конечных точек.

Если подытожить, то ресурс EndpointSlice улучшает масштабируемость Kubernetes, распределяя большое число конечных точек по разным объектам `EndpointSlice`. Это может быть полезно, если Сервисы в вашей платформе насчитывают сотни конечных точек. В зависимости от версии Kubernetes вам, вероятно, придется включить возможности EndpointSlice вручную. Если у вас Kubernetes v1.18, то нужно указать флаг компонента в kube-proxy. Начиная с версии 1.19, возможности EndpointSlice включены по умолчанию.

Аспекты реализации Сервиса

До сих пор мы обсуждали Сервисы, объекты Endpoints и возможности, которые они предоставляют приложениям в кластере Kubernetes. Но как в Kubernetes реализованы Сервисы? Как это все работает?

В данном разделе мы рассмотрим разные подходы к реализации Сервисов в Kubernetes. Сначала речь пойдет об общей архитектуре kube-proxy. Затем мы проанализируем разные режимы плоскости данных этого компонента. И в завершение будут предложены альтернативы, такие как подключаемые модули CNI, способные брать на себя обязанности kube-proxy.

Kube-proxy

Kube-proxy — это агент, который выполняется на каждом узле кластера. Его основная обязанность состоит в том, чтобы сделать Сервисы доступными для Pod'ов, развернутых на локальном узле. Для этого он следит за объектами Service и Endpoints с помощью API-сервера и программирует сетевой стек Linux (например, с помощью iptables), чтобы обрабатывать пакеты соответствующим образом.



Изначально агент kube-proxy играл роль прокси-сервера между Pod'ами, развернутыми на узле, и Сервисами. Отсюда и название. Но с развитием проекта Kubernetes он превратился из прокси-сервера в нечто, больше похожее на агента узла или локальную плоскость управления.

Kube-proxy поддерживает три режима работы: userspace (прокси-сервер в пространстве пользователя), iptables и IPVS. Режим userspace встречается редко, так как iptables и IPVS являются более удачными альтернативами, поэтому следующие разделы этой главы посвящены именно им.

Kube-proxy: режим iptables

На момент написания этих строк (Kubernetes версии 1.18) kube-proxy по умолчанию использует режим iptables. Можно с уверенностью утверждать, что на сегодня этот режим наиболее распространенный среди существующих кластеров.

В этом режиме kube-proxy задействует функции преобразования сетевых адресов (англ. Network Address Translation или NAT), которые предоставляет iptables.

Сервисы ClusterIP

Для поддержки Сервисов ClusterIP kube-proxy программирует таблицу NAT в ядре Linux, что позволяет выполнять операции DNAT (Destination NAT) для пакетов, направленных к Сервисам. Правила DNAT подставляют вместо IP-адреса назначения пакета IP-адрес конечной точки Сервиса (т. е. IP-адрес Pod'а). После этого сеть обращается с пакетом так, будто он был изначально предназначен для Pod'а.

Трафик между конечными точками разных Сервисов kube-proxy распределяет с помощью нескольких цепочек iptables:

- *Цепочка Сервисов* — цепочка верхнего уровня, содержащая правила для каждого Сервиса. Каждое правило проверяет, совпадает ли IP-адрес назначения пакета с ClusterIP Сервиса. Если да, то пакет направляется во внутрисервисную цепочку.
- *Внутрисервисная цепочка* — у каждого Сервиса есть своя цепочка iptables с правилом его конечной точки. Каждое правило учитывает расширение iptables `statistic` для выбора целевой конечной точки случайным образом. Вероятность выбора каждой конечной точки составляет $1/n$, где n — число конечных точек. После выбора пакет направляется в цепочку конечной точки Сервиса.
- *Цепочка конечной точки Сервиса* — у каждой конечной точки Сервиса есть цепочка iptables, которая выполняет операцию DNAT для пакета.

В листинге 6.4 с правилами iptables показан пример Сервиса ClusterIP. Сервис называется `nginx` и имеет три конечные точки (лишние правила были опущены для краткости).

Листинг 6.4

```
$ iptables --list --table nat
Chain KUBE-SERVICES (2 references) ❶
target     prot opt source               destination
KUBE-MARK-MASQ  tcp  -- !10.244.0.0/16        10.97.85.96
    /* default/nginx: cluster IP */ tcp dpt:80
KUBE-SVC-4N57TFCL4MD7ZTDA  tcp  --  anywhere             10.97.85.96
    /* default/nginx: cluster IP */ tcp dpt:80
KUBE-NODEPORTS  all  --  anywhere             anywhere
    /* kubernetes service nodeports; NOTE: this must be the last rule in
    this chain */ ADDRTYPE match dst-type LOCAL

Chain KUBE-SVC-4N57TFCL4MD7ZTDA (1 references) ❷
target     prot opt source               destination
KUBE-SEP-VUJFIIOGYVVPH7Q4  all  --  anywhere    anywhere    /* default/nginx: */
    statistic mode random probability 0.33333333349
KUBE-SEP-Y42457KCQHG7FFWI  all  --  anywhere    anywhere    /* default/nginx: */
    statistic mode random probability 0.50000000000
KUBE-SEP-UOUQBAIW4Z676WKH  all  --  anywhere    anywhere    /* default/nginx: */
```

```
Chain KUBE-SEP-UOUQBAIW4Z676WKH (1 references) ❸
target     prot opt source              destination
KUBE-MARK-MASQ  all  --  10.244.0.8   anywhere             /* default/nginx: */
DNAT       tcp  --  anywhere          anywhere             /* default/nginx: */
    tcp to:10.244.0.8:80

Chain KUBE-SEP-VUJFIIOGYVVPH7Q4 (1 references)
target     prot opt source              destination
KUBE-MARK-MASQ  all  --  10.244.0.108   anywhere           /* default/nginx: */
DNAT       tcp  --  anywhere            anywhere           /* default/nginx: */
    tcp to:10.244.0.108:80

Chain KUBE-SEP-Y42457KCQHG7FFWI (1 references)
target     prot opt source              destination
KUBE-MARK-MASQ  all  --  10.244.0.6         anywhere       /* default/nginx: */
DNAT       tcp  --  anywhere            anywhere           /* default/nginx: */
    tcp to:10.244.0.6:80
```

❶ Это цепочка верхнего уровня. Она содержит правила для всех Сервисов в кластере. Обратите внимание, что KUBE-SVC-4N57TFCL4MD7ZTDA задает IP-адрес назначения 10.97.85.96. Это ClusterIP Сервиса nginx.

❷ Это цепочка Сервиса nginx. Обратите внимание на то, что у каждой конечной точки Сервиса есть свое правило и вероятность, с которой оно будет выбрано.

❸ Эта цепочка относится к одной из конечных точек Сервиса (SEP означает Service endpoint). Последнее правило в этой цепочке выполняет операцию DNAT, чтобы направить пакет к конечной точке (или Pod'у).

Сервисы NodePort и LoadBalancer

Для Сервисов NodePort и LoadBalancer kube-proxy конфигурирует правила, похожие на те, которые используются для Сервисов ClusterIP. Главное отличие в том, что эти правила сопоставляют пакеты по номеру их порта назначения. Если номер совпадает, правило направляет пакет во внутрисервисную цепочку, где выполняется операция DNAT. В листинге 6.5 показаны правила iptables для Сервиса NodePort под названием nginx, который прослушивает порт 31767.

Листинг 6.5

```
$ iptables --list --table nat
Chain KUBE-NODEPORTS (1 references) ❶
target     prot opt source              destination
KUBE-MARK-MASQ  tcp  --  anywhere             anywhere       /* default/nginx: */
    tcp dpt:31767
KUBE-SVC-4N57TFCL4MD7ZTDA  tcp  --  anywhere    anywhere     /* default/nginx: */
    tcp dpt:31767 ❷
```

❶ Kube-proxy программирует правила iptables для Сервисов NodePort в цепочке KUBE-NODEPORTS.

❷ Если в качестве порта назначения пакета указано `tcp: 31767`, то он направляется во внутрисервисную цепочку. Это та цепочка, которую мы видели в пункте 2 предыдущего листинга 6.4.

Помимо программирования правил iptables, kube-proxy открывает порт, назначенный Сервису NodePort, и поддерживает его в открытом состоянии. Данные действия всего лишь не дают другим процессам занять этот порт, они не имеют никакого отношения к маршрутизации.

Ключевой фактор, на который следует обращать внимание при использовании Сервисов NodePort и LoadBalancer, — параметр политики для внешнего трафика. Он определяет, куда Сервис направляет внешний трафик: к конечным точкам отдельных узлов (`externalTrafficPolicy: Local`) или всего кластера (`externalTrafficPolicy: Cluster`). Как вы увидите далее, у каждой из этих политик есть свои преимущества и недостатки.

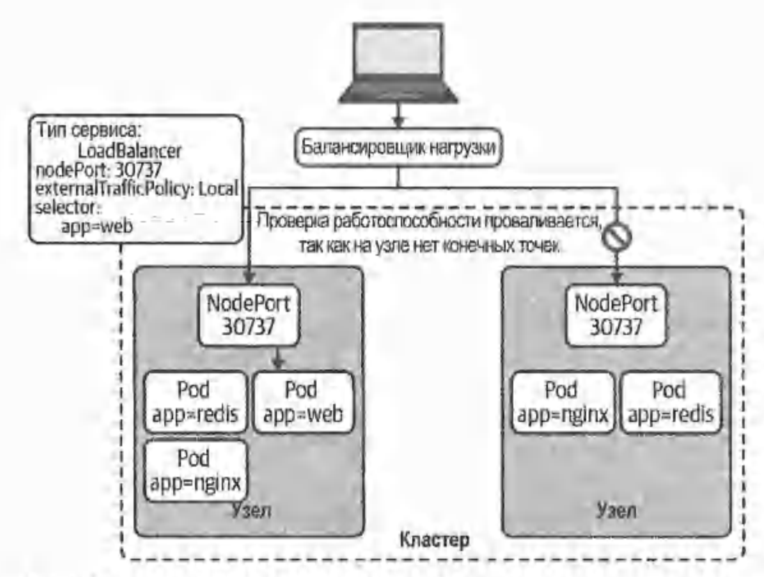


Рис. 6.6. Сервис LoadBalancer с политикой `Local` для внешнего трафика. Внешний балансировщик нагрузки выполняет проверки работоспособности узла. Любой узел, у которого нет конечной точки Сервиса, удаляется из внутреннего пула балансировщика

Когда указана политика `Local`, Сервис направляет трафик к конечным точкам (Pod'ам) того узла, который этот трафик принял. Направление пакетов к локальным конечным точкам имеет два важных преимущества. Первое, в этом процессе не участвует операция `SNAT`, что сохраняет исходный IP-адрес и делает его доступным для приложений. И второе — отсутствует лишний сетевой переход, который пришлось бы выполнить при перенаправлении трафика к другому узлу. Тем не менее, политика `Local` имеет свои недостатки. Главный из них состоит в том, что трафик, дошедший до узла, на котором нет конечных точек Сервиса, отклоняется. В связи с этим политика `Local` обычно используется в сочетании с внешним балансировщиком нагрузки, который проверяет работоспособность узлов. Если у узла нет конеч-

ных точек Сервиса, балансировщик нагрузки не направляет ему трафик, так как он не может пройти проверку работоспособности. Эта функциональность проиллюстрирована на рис. 6.6. Еще один недостаток политики Local — вероятность несбалансированной нагрузки. Например, если у узла есть три конечные точки Сервиса, каждая из них получит 33 % трафика. Если же конечная точка присутствует на узле в единственном экземпляре, она получит весь трафик. Этот перекос можно смягчить, распределяя Pod'ы с помощью правил антиподобия или за счет использования DaemonSet для планирования работы Pod'ов.

Если у вас есть Сервис, который обрабатывает огромные объемы внешнего трафика, то выбор политики `Local` обычно себя оправдывает. Но, если в вашем распоряжении нет балансировщика нагрузки, следует предпочесть политику `Cluster`. Она делает так, что внешний трафик распределяется между всеми конечными точками кластера, как показано на рис. 6.7. Как можно догадаться, балансировка нагрузки приводит к потере исходного IP-адреса ввиду применения SNAT. Это также может вылиться в дополнительный сетевой переход. С другой стороны, политика `Cluster` не отбрасывает внешний трафик, независимо от того, где размещены Pod'ы конечных точек.

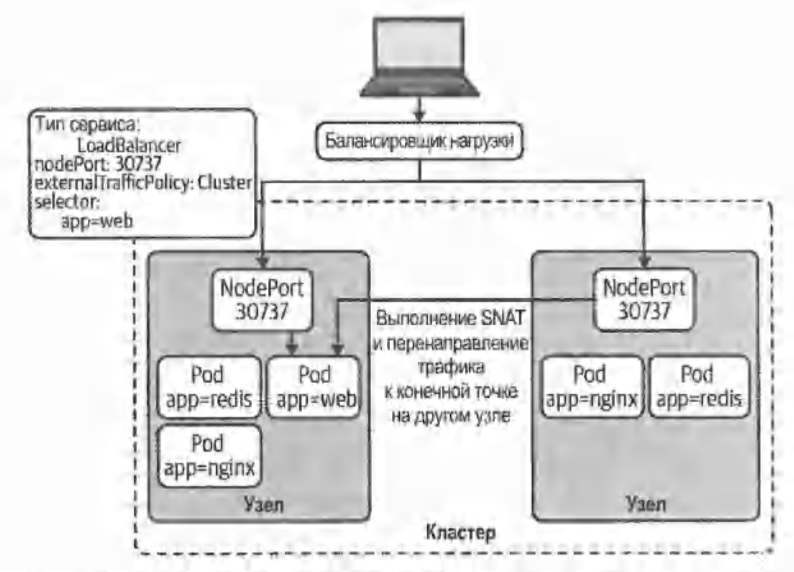


Рис. 6.7. Сервис LoadBalancer с политикой `Cluster` для внешнего трафика. Узлы, у которых отсутствуют локальные конечные точки, перенаправляют трафик к конечным точкам на других узлах

Отслеживание соединений

Когда сетевой стек ядра выполняет операцию DNAT для пакета, предназначенного для Сервиса, он добавляет запись в таблицу отслеживания соединений (англ. connection tracking или conntrack). Эта таблица хранит сведения о проведенных преобразованиях, чтобы их можно было применять ко всем последующим пакетам,

направленным к тому же Сервису. Она также удаляет NAT из ответных пакетов перед их отправкой исходному Pod'у.

Каждая запись в таблице связывает между собой протокол, исходный IP-адрес, исходный порт, IP-адрес назначения и порт назначения, используемые до NAT, с аналогичными параметрами, но уже после NAT (эти записи содержат дополнительную информацию, но в данном контексте она неважна). На рис. 6.8 изображена табличная запись, которая представляет соединение, исходящее от Pod'а (`192.168.0.9`) к Сервису (`10.96.0.14`). Обратите внимание, что после применения DNAT IP-адрес и порт назначения меняются.



Рис. 6.8. Табличная запись для отслеживания соединения (conntrack) от Pod'а (`192.168.0.9`) к Сервису (`10.96.0.14`)



Когда таблица conntrack становится полной, ядро начинает разрывать или отклонять соединения, что может создать проблемы для некоторых приложений. Если это происходит с вашими приложениями, которые принимают много соединений, вам, возможно, следует отрегулировать максимальный размер conntrack на ваших узлах. Что еще более важно, вы должны организовать мониторинг свободного пространства в этой таблице, чтобы получать предупреждения в случае, если оно заканчивается.

Маскарадинг

Вы могли заметить, что в предыдущем примере мы обошли вниманием правила iptables `KUBE-MARK-MASQ`, предназначенные для пакетов, которые узел принимает из-за пределов кластера. Чтобы как следует маршрутизировать эти пакеты, механизм управления Сервисами должен маскировать пакеты, направляемые к другим узлам (или применять к ним операцию замены источника [англ. source NAT]). В противном случае ответный пакет будет содержать IP-адрес Pod'а, которая обработала запрос. Это вызовет проблемы, так как клиент инициировал соединение с узлом, а не с Pod'ом.

Маскарадинг также применяется для вывода трафика из кластера. Когда Pod подключается к внешним сервисам, исходный IP-адрес должен принадлежать не ей, а узлу, на котором она размещена, иначе сеть отклонит ответные пакеты, так как IP-адрес назначения не будет принадлежать этому Pod'у.

Вопросы, связанные с производительностью

Режим iptables, как и прежде, продолжает служить верой и правдой кластерам Kubernetes. Тем не менее, вы должны понимать, что ему присущи некоторые огра-

ничения, связанные с производительностью и масштабированием, так как вы можете с ними столкнуться в крупных кластерах.

Учитывая структуру правил iptables и то, как они работают, каждый раз, когда Pod устанавливает новое соединение с Сервисом, начальный пакет проходит по всем этим правилам, пока не найдет подходящее. В худшем случае ему придется перебрать весь набор правил iptables.

Алгоритм обработки пакетов в режиме iptables характеризуется временной сложностью O(n). Иными словами, этот режим масштабируется линейно при увеличении количества Сервисов в кластере. И чем больше Сервисов, тем медленней происходит соединение с ними.

Наверное, самым важным аспектом является то, что аналогичные проблемы в крупномасштабных кластерах присущи и процессу обновления правил iptables. Поскольку эти правила не инкрементные, агенту kube-proxy приходится перезаписывать всю таблицу при каждом обновлении. В некоторых случаях это может занять не одну минуту, из-за чего возникает риск передачи трафика выведенным из эксплуатации конечным точкам. Более того, во время этих обновлений агент kube-proxy должен удерживать блокировку iptables (**/run/xtables.lock**), что может привести к соперничеству за ресурсы с другими процессами, которым нужно обновлять правила iptables, такими как подключаемые модули CNI.

Линейное масштабирование — нежелательное свойство для любой системы. С другой стороны, если верить тестам, проведенным сообществом Kubernetes (**https://oreil.ly/YJAu9**), вы не должны заметить никакого ухудшения производительности, если только ваш кластер не состоит из десятков тысяч Сервисов. Но, если вы имеете дело с такими масштабами, вам может пригодиться режим kube-proxy IPVS, который мы обсудим в следующем разделе.

Плавающие обновления и цикл согласования Сервисов

Одна интересная проблема сервисов заключается в неожиданных ошибках запросов при выполнении плавающих обновлений приложения. В окружениях для разработки она встречается не так часто, но вы можете столкнуться с ней в кластерах в процессе эксплуатации, в которых развернуто множество приложений.

Суть проблемы состоит в распределенности Kubernetes. Как уже обсуждалось в этой главе, работа Сервисов в Kubernetes основана на взаимодействии нескольких компонентов, главными из которых являются контроллер Endpoints и kube-proxy.

Вместе с удалением Pod'а происходит следующее:

- Агент kubelet инициирует процедуру выключения Pod'а. Он шлет приложению сигнал SIGTERM и ждет, когда оно остановится. Если процесс продолжает работать по окончании периода, предусмотренного для его остановки, kubelet направляет ему сигнал принудительного завершения, SIGKILL.
- Контроллер Endpoints получает событие об уничтожении Pod'а, что влечет за собой удаление ее IP-адреса из ресурса Endpoints. После обновления этого ресурса kube-proxy убирает тот же IP-адрес из списка правил iptables (или виртуального сервиса IPVS).

> Это классическое состояние гонки в распределенной системе. В идеале контроллер Endpoints и kube-proxy успевают выполнить свои обновления еще до завершения работы Pod'а. Но порядок выполнения операций не гарантируется, так как они работают параллельно. Есть вероятность того, что приложение завершится (и, следовательно, прекратит прием запросов) до того, как kube-proxy на каждом узле удалит Pod из списка активных конечных точек. Если это произойдет, уже посланные запросы окажутся неудачными, так как они направляются к Pod'у, который больше не работает.
>
> Чтобы решить эту проблему, платформа Kubernetes должна подождать, пока kube-proxy не завершит обновление конечных точек, и только потом приступать к остановке приложений. Но это непрактично. Что бы вы, к примеру, делали в ситуации, когда узел становится недоступным? Тем не менее, на практике мы смягчали эту проблему с помощью обработчиков SIGTERM и операций sleep в хуках, предшествующих остановке.

Kube-proxy: режим IPVS

IPVS (IP Virtual Server — виртуальный IP-сервер) — механизм балансировки нагрузки, встроенный в ядро Linux. Поддержка IPVS была добавлена в kube-proxy в качестве ответа на ограничения масштабирования и проблемы с производительностью, присущие режиму iptables.

Как уже обсуждалось в предыдущем разделе, в режиме iptables для реализации Сервисов Kubernetes используются правила iptables. Эти правила имеют вид списка, который в худшем случае пакетам нужно перебирать целиком. У режима IPVS такой проблемы нет, так как он был изначально предназначен для задач, которым требуется балансировка нагрузки.

Реализация IPVS в ядре Linux ищет путь назначения пакета с помощью хеш-таблиц. Пи установлении нового соединения IPVS не перебирает список Сервисов, а сразу же находит Pod, которому направлен пакет. Для этого служит IP-адрес заданного Сервиса.

Давайте поговорим о том, как kube-proxy в режиме IPVS работает с каждым отдельным типом Сервисов Kubernetes.

Сервисы ClusterIP

При работе с Сервисами, у которых есть ClusterIP, агент kube-proxy в режиме `ipvs` делает несколько вещей. Прежде всего, он назначает IP-адрес Сервиса ClusterIP фиктивному сетевому интерфейсу узла под названием `kube-ipvs0`, как показано в листинге 6.6.

Листинг 6.6

```
$ ip address show dev kube-ipvs0
28: kube-ipvs0: <BROADCAST,NOARP> mtu 1500 qdisc noop state DOWN group default
    link/ether 96:96:1b:36:32:de brd ff:ff:ff:ff:ff:ff
    inet 10.110.34.183/32 brd 10.110.34.183 scope global kube-ipvs0
       valid_lft forever preferred_lft forever
```

```
    inet 10.96.0.10/32 brd 10.96.0.10 scope global kube-ipvs0
       valid_lft forever preferred_lft forever
    inet 10.96.0.1/32 brd 10.96.0.1 scope global kube-ipvs0
       valid_lft forever preferred_lft forever
```

После обновления фиктивного интерфейса kube-proxy создает виртуальный сервис IPVS с тем же IP-адресом, что и у Сервиса ClusterIP. И в завершение для каждой конечной точки Сервиса в такой виртуальный сервис добавляется реальный сервер IPVS. В листинге 6.7 показаны виртуальный сервис и реальные серверы IPVS для Сервиса ClusterIP с тремя конечными точками.

Листинг 6.7

```
$ ipvsadm --list --numeric --tcp-service 10.110.34.183:80
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  10.110.34.183:80 rr ❶
  -> 192.168.89.153:80            Masq    1      0          0 ❷
  -> 192.168.89.154:80            Masq    1      0          0
  -> 192.168.89.155:80            Masq    1      0          0
```

❶ Это виртуальный сервис IPVS. Он имеет тот же IP-адрес, что и Сервис ClusterIP.

❷ Это один из реальных серверов IPVS. Он соответствует одной из конечных точек Сервиса (Pod'a).

Сервисы NodePort и LoadBalancer

В случае с Сервисами NodePort и LoadBalancer kube-proxy создает виртуальный сервис IPVS для соответствующего ClusterIP. То же самое kube-proxy делает для каждого IP-адреса кольцевого адреса узла. Например, в листинге 6.8 приведен список виртуальных сервисов IPVS, созданных для Сервиса NodePort, который прослушивает TCP-порт 30737.

Листинг 6.8

```
ipvsadm --list --numeric
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port       Forward Weight ActiveConn InActConn
TCP  10.0.99.67:30737 rr ❶
  -> 192.168.89.153:80        Masq    1      0          0
  -> 192.168.89.154:80        Masq    1      0          0
  -> 192.168.89.155:80        Masq    1      0          0
TCP  10.110.34.183:80 rr ❷
  -> 192.168.89.153:80        Masq    1      0          0
  -> 192.168.89.154:80        Masq    1      0          0
  -> 192.168.89.155:80        Masq    1      0          0
```

```
TCP  127.0.0.1:30737 rr ❸
  -> 192.168.89.153:80       Masq    1      0          0
  -> 192.168.89.154:80       Masq    1      0          0
  -> 192.168.89.155:80       Masq    1      0          0
TCP  192.168.246.64:30737 rr ❹
  -> 192.168.89.153:80       Masq    1      0          0
  -> 192.168.89.154:80       Masq    1      0          0
  -> 192.168.89.155:80       Masq    1      0          0
```

❶ Виртуальный сервис IPVS принимает соединения на IP-адрес узла.

❷ Виртуальный сервис IPVS принимает соединения на IP-адрес Сервиса.

❸ Виртуальный сервис IPVS принимает соединения на **localhost**.

❹ Виртуальный сервис IPVS принимает соединения на вторичный сетевой интерфейс узла.

Работа без kube-proxy

Так сложилось, что агент kube-proxy стал неотъемлемой частью любого кластера Kubernetes. Это виртуальный компонент, который делает возможным работу Сервисов Kubernetes. Но сообщество разработчиков не стоит на месте, и, может быть, в будущем некоторые кластеры смогут обходиться без kube-proxy. Как же это возможно? Как будет происходить управление Сервисами?

С изобретением eBPF (extended Berkeley Packet Filters — пакет расширенных фильтров Berkley) подключаемые модули CNI, такие как Cilium (**https://oreil.ly/sWoh5**) и Calico (**https://oreil.ly/0jrKG**) могут брать на себя обязанности kube-proxy. Вместо того чтобы управлять Сервисами с помощью iptables или IPVS, подключаемые модули CNI программируют Сервисы прямо в плоскость данных Pod'a. Использование eBPF улучшает производительность и масштабируемость Сервисов Kubernetes, так как реализация этой технологии ищет конечные точки с помощью хеш-таблиц. Это также положительно сказывается на обработке обновлений Сервисов, ведь eBPF может эффективно обрабатывать каждое внесенное изменение по отдельности.

Отсутствие необходимости в kube-proxy и более эффективная маршрутизация Сервисов — это достойные цели, особенно для тех, кто имеет дело с крупномасштабными системами. Однако эти решения еще не готовы к применению в промышленных условиях. Например, для поддержки развертывания без участия kube-proxy реализации Cilium нужна более новая версия ядра (на момент написания этих строк последней версией Cilium является 1.8). Аналогичным образом команда разработчиков Calico не рекомендует внедрять технологию eBPF в промышленных системах, так как ее поддержка находится на стадии предварительного тестирования (на момент написания этих строк последней версией Calico является 3.15.1). Мы ожидаем, что со временем альтернативы kube-proxy станут более распространенными. Cilium даже поддерживает использование своего прокси-сервера вместе с другими подключаемыми модулями CNI (так называемое построение цепочки CNI; см. **https://oreil.ly/jZ-2r**).

Обнаружение сервисов

Механизм обнаружения сервисов дает возможность приложениям находить сервисы, доступные в сети. И хотя это не вопрос *маршрутизации*, обнаружение сервисов имеет непосредственное отношение к Сервисам Kubernetes.

Разработчики платформы могут задаться вопросом, стоит ли им внедрять в кластер отдельную систему обнаружения сервисов, такую как Consul. Такое решение возможно, но не обязательно, так как Kubernetes предоставляет аналогичные возможности всем приложениям, выполняемым в кластере. В данном разделе мы обсудим различные механизмы обнаружения сервисов, доступные в Kubernetes и основанные на DNS, API-интерфейсе и переменных окружения.

Использование DNS

Приложения, работающие внутри кластера Kubernetes, могут обнаруживать сервисы по DNS. Соответствующие развертывания содержат DNS-серверы, интегрированные с API-интерфейсом Kubernetes. На сегодня чаще всего для этого используется расширяемый DNS-сервер с открытым исходным кодом под названием CoreDNS (**https://coredns.io**).

CoreDNS следит за ресурсами в API-интерфейсе Kubernetes. Для каждого Сервиса он создает DNS-запись в следующем формате: `<имя-сервиса>.<название-пространства-имен>.svc.cluster.local`. Например, Сервис под названием `nginx` в пространстве имен `default` получит DNS-запись `nginx.default.svc.cluster.local`. Но как эти записи могут использовать Pod'ы?

Чтобы включить обнаружение сервисов на основе DNS, Kubernetes указывает CoreDNS в качестве средства поиска IP-адресов для Pod'ов. Во время конфигурации изолированной среды для Pod'a kubelet создает файл **/etc/resolv.conf**, назначая CoreDNS сервером доменных имен, и внедряет этот файл в контейнер. Содержимое **/etc/resolv.conf** Pod'a выглядит примерно так:

```
$ cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
```

Учитывая эту конфигурацию, каждый раз, когда Pod'y нужно соединиться с Сервисом по его имени, она направляет DNS-запросы серверу CoreDNS.

Еще один интересный прием в конфигурации сопоставителя доменных имен состоит в указании параметров `ndots` и `search` для упрощения DNS-запросов. Если Pod'y нужно обратиться к Сервису, размещенному в том же пространстве имен, она может сделать это по его названию, не используя полное доменное имя (`nginx.default.svc.cluster.local`):

```
$ nslookup nginx
Server:         10.96.0.10
Address:        10.96.0.10#53
Name:   nginx.default.svc.cluster.local
Address: 10.110.34.183
```

Аналогичным образом, когда Pod хочет соединиться с Сервисом в другом пространстве имен, она может добавить название этого пространства к имени Сервиса:

```
$ nslookup nginx.default
Server:         10.96.0.10
Address:        10.96.0.10#53
Name:   nginx.default.svc.cluster.local
Address: 10.110.34.183
```

При использовании конфигурации `ndots` следует обратить внимание на то, как она влияет на приложения, которые взаимодействуют с сервисами из-за пределов кластера. Параметр `ndots` определяет, сколько точек должно содержаться в имени, чтобы оно считалось полным или абсолютным. При сопоставлении неполного имени система пытается выполнить различные поисковые запросы, используя элементы параметра `search`, как видно в следующем примере. Таким образом, при сопоставлении неполных имен, принадлежащих сервисам за пределами кластера, сопоставитель сверяется с кластерным DNS-сервером, направляя ему несколько тщетных запросов, и только затем начинает считать это имя абсолютным. Чтобы избежать этой проблемы, можно задать в своих приложениях полные доменные имена с точкой в конце. В качестве альтернативной меры можно откорректировать DNS-конфигурацию Pod'а, изменив поле `dnsConfig` в ее спецификации.

В листинге 6.9 показано, как конфигурация `ndots` влияет на Pod'ы, которые сопоставляют внешние имена. Обратите внимание, что сопоставление имени, число точек в котором меньше, чем указано в `ndots`, требует нескольких DNS-запросов, в то время как для сопоставления абсолютного имени нужен всего один запрос.

Листинг 6.9

```
$ nslookup -type=A google.com -debug | grep QUESTIONS -A1 ❶
  QUESTIONS:
google.com.default.svc.cluster.local, type = A, class = IN
--
  QUESTIONS:
google.com.svc.cluster.local, type = A, class = IN
--
  QUESTIONS:
google.com.cluster.local, type = A, class = IN
--
  QUESTIONS:
google.com, type = A, class = IN

$ nslookup -type=A -debug google.com. | grep QUESTIONS -A1 ❷
  QUESTIONS:
google.com, type = A, class = IN
```

❶ Попытка сопоставления имени с менее чем пятью точками (что делает его неполным). Сопоставитель выполняет несколько поисковых запросов, по одному для каждого элемента в поле `search` в файле **/etc/resolv.conf**.

❷ Попытка сопоставить полное доменное имя. Сопоставитель выполняет один запрос.

В целом обнаружение сервисов по DNS — чрезвычайно полезный механизм, который упрощает взаимодействие приложений и Сервисов Kubernetes.

Использование API-интерфейса Kubernetes

Еще один метод обнаружения Сервисов состоит в использовании API-интерфейса Kubernetes. Сообщество Kubernetes поддерживает различные клиентские библиотеки, написанные на разных языках, включая Go, Java, Python и др. Некоторые платформы для разработки приложений такие, как Spring, тоже поддерживают обнаружение сервисов посредством Kubernetes API.

Применение API-интерфейса Kubernetes для обнаружения сервисов может быть полезным в некоторых ситуациях. Например, если вашему приложению нужно знать об изменении конечных точек Сервиса в момент, когда оно происходит, отслеживание API-интерфейса придется кстати.

Главный недостаток организации обнаружения сервисов на основе API-интерфейса Kubernetes заключается в том, что ваше приложение жестко привязывается к платформе. В идеале приложения не должны ничего знать о том, где они выполняются. Если вы все же решите задействовать Kubernetes API для обнаружения сервисов, то можете разработать интерфейс, который будет скрывать подробности работы Kubernetes от вашей бизнес-логики.

Использование переменных окружения

Для организации процесса обнаружения сервисов Kubernetes в каждый Pod добавляются переменные окружения. Переменные, которые устанавливаются каждому Сервису, основаны на его определении. Например, переменные окружения для Сервиса ClusterIP `nginx`, прослушивающего порт 80, выглядят так:

```
NGINX_PORT_80_TCP_PORT=80
NGINX_SERVICE_HOST=10.110.34.183
NGINX_PORT=tcp://10.110.34.183:80
NGINX_PORT_80_TCP=tcp://10.110.34.183:80
NGINX_PORT_80_TCP_PROTO=tcp
NGINX_SERVICE_PORT=80
NGINX_PORT_80_TCP_ADDR=10.110.34.183
```

Недостаток этого подхода состоит в том, что переменные окружения нельзя обновить без перезапуска Pod'а, поэтому Сервис должен быть сконфигурирован до запуска Pod'а.

Производительность DNS-сервиса

Как уже упоминалось в предыдущем разделе, предоставление приложениям механизма обнаружения сервисов на основе DNS является важным аспектом вашей платформы. По мере увеличения вашего кластера и количества приложений DNS-сервис может перестать справляться с нагрузкой. В этом разделе мы обсудим методики, с помощью которых вы можете сделать свой DNS-сервис высокопроизводительным.

DNS-кэш на каждом узле

Сообщество Kubernetes занимается развитием дополнения под названием NodeLocal DNSCache (**https://oreil.ly/lQdTH**), которое кэширует DNS-запросы на каждом узле, что позволяет решить сразу несколько проблем. Во-первых, кэш уменьшает латентность DNS-запросов, так как ваши приложения получают свои ответы из локального хранилища (при условии, что там есть нужная запись) вместо того, чтобы обращаться к DNS-серверу (который может находиться на другом узле). Во-вторых, снижается нагрузка на серверы CoreDNS, так как приложения в большинстве случаев пользуются кэшем. Наконец, если происходит промах кэша, механизм кэширования преобразует DNS-запрос в TCP-соединение при обращении к центральному DNS-сервису. Использование TCP вместо UDP делает DNS-запросы более надежными.

DNS-кэш кластера представляет собой объект `DaemonSet`. Каждая его реплика перехватывает DNS-запросы, исходящие из того узла, на котором она находится. Для использования кэша не нужно модифицировать код или конфигурацию приложения. Архитектура NodeLocal DNSCache на уровне узла изображена на рис. 6.9.

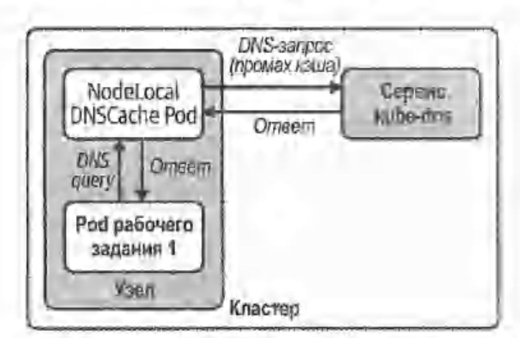


Рис. 6.9 Архитектура дополнения NodeLocal DNSCache на уровне узла. DNS-кэш перехватывает DNS-запросы и в случае попадания кэша немедленно отвечает. Если случился промах кэша, запрос направляется к DNS-сервису кластера

Автомасштабирование Развертывания DNS-сервера

В дополнение к локальным копиям DNS-кэша на каждом отдельном узле вы можете автоматически развертывать соответствующий объект Развертывания в зависимости от размера кластера. Следует отметить, что эта стратегия не подразумевает наличия средства горизонтального автомасштабирования Pod'ов. Вместо этого она применяет средство пропорционального автомасштабирования (**https://oreil.ly/432wc**), которое привязывает число приложений к числу узлов в кластере.

Средство автомасштабирования кластера также является Pod'ом. Оно предоставляет флаг конфигурации, с помощью которого можно указать приложения, нуждающиеся в автоматическом масштабировании. Чтобы масштабировать DNS, необходимо установить целевой флаг для Развертывания CoreDNS (или kube-dns). После запуска средство автомасштабирования начинает регулярно обращаться к API-серверу

(по умолчанию раз в 10 минут), чтобы узнать, сколько узлов и процессорных ядер имеется в кластере. Затем оно при необходимости корректирует число реплик в Развертывании CoreDNS. Желаемое число реплик определяется соотношением реплики/узлы или реплики/ядра, которое можно настраивать. Подходящее соотношение зависит от ваших приложений и от того, насколько активно они используют DNS.

Ingress

Как уже обсуждалось в *главе 5*, приложения, выполняемые в Kubernetes, обычно недоступны из-за пределов кластера. Если у вашего приложения отсутствуют внешние клиенты, в этом нет никакой проблемы. Отличный пример таких приложений — пакетные обработки. Однако в реальности Kubernetes в большинстве случаев предназначен для развертывания веб-сервисов, у которых есть конечные пользователи.

Ingress — это метод предоставления клиентам, находящимся за пределами кластера, доступа к сервисам, размещенным в Kubernetes. Несмотря на то, что API-интерфейс Ingress не реализован в стандартной конфигурации, он является неотъемлемой частью любой платформы, основанной на Kubernetes. Готовые приложения и расширения нередко рассчитывают на то, что в вашем кластере есть контроллер Ingress. Более того, без него ваши разработчики не смогут успешно выполнять свои приложения в Kubernetes.

В этом разделе вы познакомитесь с факторами, которые вам необходимо учитывать при реализации Ingress в вашей платформе. Мы проведем обзор API-интерфейса Ingress, самых распространенных методов обработки входящего трафика, с которыми вы будете сталкиваться, и поговорим о важной роли контроллеров Ingress в работе платформы, основанной на Kubernetes. Мы также обсудим разные способы развертывания этих контроллеров и их недостатки. В заключение будут рассмотрены распространенные проблемы и полезные средства интеграции с другими инструментами в экосистеме Kubernetes.

Зачем нужен механизм Ingress

Трафик к Pod'ам можно направлять с помощью Сервисов Kubernetes, так зачем же нам еще одна стратегия, которая делает то же самое? Несмотря на то, что мы стремимся сделать наши платформы простыми, реальность такова, что Сервисы обладают существенными ограничениями и недостатками:

- *Ограниченные возможности маршрутизации.* Сервисы маршрутизируют трафик в соответствии с IP-адресом и портом назначения входящих запросов. Это может быть удобно в небольших и относительно простых приложениях, но в более крупных проектах, основанных на микросервисах, такой подход быстро теряет свою привлекательность. Такого рода проекты требуют более гибкие механизмы маршрутизации и другие нетривиальные возможности.

- *Цена*. Если ваш кластер размещен в облачном окружении, каждый его Сервис LoadBalancer создает внешний балансировщик нагрузки, такой как ELB в случае с AWS. Наличие отдельного балансировщика нагрузки для каждого Сервиса в вашей платформе может быстро сделать ваши расходы непомерными.

Механизм Ingress позволяет избавиться от этих двух ограничений. Его возможности балансировки нагрузки и маршрутизации работают не на 3–4 уровнях модели OSI (сетевой и транспортный уровни), а на прикладном уровне (L7), что делает процесс более эффективным.

Еще одно преимущество этого механизма состоит в том, что он устраняет необходимость во множестве балансировщиков нагрузки или точек входа в платформу. Благодаря развитым средствам маршрутизации, доступным в Ingress, таким, как возможность направлять HTTP-запросы с учетом заголовка `Host`, трафик всех сервисов может поступать в кластер через одну-единственную точку входа, а его демультиплексирование возьмет на себя контроллер Ingress. Это кардинально удешевляет процесс обработки трафика в вашей платформе.

Наличие единой точки входа в платформу также упрощает внеоблачные развертывания. Вместо того чтобы управлять множеством внешних баласировщиков нагрузки с помощью большого числа Сервисов NodePort, вы можете использовать всего один внешний балансировщик, который направляет трафик к контроллерам Ingress.

Несмотря на то, что Ingress решает большинство проблем, связанных с Сервисами Kubernetes, необходимость в последних по-прежнему присутствует. Контроллеры Ingress сами по себе работают внутри платформы, поэтому к ним нужно открывать доступ для клиентов, которые находятся снаружи. И для этого подойдет как раз Сервис (либо NodePort, либо LoadBalancer). Кроме того, сильной стороной большинства контроллеров Ingress является балансировка HTTP-трафика. Если вы хотите иметь возможность развертывать приложения с поддержкой других протоколов, вам придется сочетать Ingress вместе с Сервисами (смотря какими возможностями обладает ваш контроллер Ingress).

API-интерфейс Ingress

API-интерфейс Ingress позволяет разработчикам приложений делать свои сервисы доступными и конфигурировать маршрутизацию запросов в соответствии со своими нуждами. Поскольку Ingress делает акцент на маршрутизацию HTTP-трафика, ресурс Ingress позволяет маршрутизировать трафик с учетом различных свойств входящих HTTP-запросов.

Один из распространенных методов маршрутизации трафика основан на HTTP-заголовке `Host`. Например, в следующем примере Ingress HTTP-запросы, у которых заголовок `Host` равен `bookhotels.com`, направляются к одному сервису, а запросы, предназначенные для `bookflights.com`, к другому (листинг 6.10).

Листинг 6.10

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hotels-ingress
spec:
  rules:
  - host: bookhotels.com
    http:
      paths:
      - path: /
        backend:
          serviceName: hotels
          servicePort: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: flights-ingress
spec:
  rules:
  - host: bookflights.com
    http:
      paths:
      - path: /
        backend:
          serviceName: flights
          servicePort: 80
```

Размещение приложений в определенных поддоменах общекластерного доменного имени — распространенный подход, который мы часто встречаем в Kubernetes. В этом случае платформе назначается доменное имя, а каждое приложение получает отдельный поддомен. Развивая тему путешествий из предыдущего примера, приложения для бронирования номеров в отелях и авиабилетов могли бы иметь адреса **hotels.cluster1.useast.example.com** и **flights.cluster1.useast.example.com**. Маршрутизация на основе поддоменов — это одна из лучших стратегий, которые вам доступны. Она также делает возможными другие интересные сценарии использования, например, предоставление клиентам приложения SaaS (software-as-a-service — программное обеспечение как услуга) отдельных доменных имен (например, **tenantA.example.com** и **tenantB.example.com**). Подробней о том, как реализовать маршрутизацию на основе поддоменов, мы поговорим в одном из следующих разделов.

Конфликты конфигурации Ingress и как их избежать

В мультитенантных и многопользовательских кластерах API-интерфейсу Ingress свойственны конфликты конфигурации. Показательным примером является ситуация, когда разные группы разработчиков пытаются использовать одно и то же доменное имя для предоставления доступа к своим приложениям. Представьте, что разработчики приложения создали ресурс Ingress с хостом `app.bearcanoe.com`. Что произойдет, если другая команда создаст Ingress с тем же хостом? API-интерфейс Ingress не уточняет, что делать в такой ситуации. Решение принимает контроллер Ingress. Некоторые контроллеры выполняют слияние конфигурации, если это возможно, а другие просто отклоняют новый ресурс Ingress и записывают в журнал сообщение об ошибке. В любом случае дублирование конфигурации в ресурсах Ingress может привести к неожиданному поведению и даже сбоям!

Обычно мы решаем эту проблему одним из двух способов. Первый способ состоит в использовании контроллера допуска, который проверяет входящий ресурс Ingress и следит за тем, чтобы имя хоста было уникальным в пределах всего кластера. За время работы в этой области мы написали множество таких контроллеров. В настоящее время для решения этого вопроса мы применяем Open Policy Agent (OPA). Сообщество OPA даже предусмотрело на этот случай отдельную политику (**https://oreil.ly/wnN0V**).

Контроллер Ingress под названием Contour предлагает другое решение данной проблемы. Если коротко, то мы сначала указываем хост в *корневом* пользовательском ресурсе HTTPProxy, а затем *подключаем* к нему другие ресурсы HTTPProxy, размещенные в рамках этого домена. Идея состоит в том, что оператор управляет корневыми ресурсами HTTPProxy и распределяет их между разными группами разработчиков. Например, оператор может создать корневой ресурс HTTPProxy с хостом **app1.bearcanoe.com** и *включить* все ресурсы HTTPProxy в пространство имен `app1`. Подробности можно найти в документации Contour (**https://oreil.ly/xOzBF**).

Возможности API-интерфейса Ingress не ограничиваются одной лишь маршрутизацией на основе хостов. В процессе развития проекта Kubernetes данный интерфейс был расширен с помощью контроллеров Ingress. К сожалению, это было сделано с использованием аннотаций, а не за счет развития ресурса Ingress. Проблема аннотаций состоит в отсутствии четкой структуры. Это может создать трудности для тех, кто их применяет, так как у API-сервера нет возможности обнаружить некорректную конфигурацию. В качестве решения некоторые контроллеры Ingress предоставляют определения пользовательских ресурсов (англ. Custom Resource Definitions или CRD). Они имеют четко определенные API-интерфейсы и поддерживают возможности, отсутствующие у механизма Ingress. Contour, к примеру, предоставляет пользовательский ресурс `HTTPProxy`. Эти CRD обеспечивают вам доступ к более широкому спектру возможностей, но при этом вы теряете способность поменять один контроллер Ingress на другой. Иными словами, вам придется пользоваться тем контроллером, который вы однажды выбрали.

Контроллеры Ingress и принцип их работы

Вспомните свои первые эксперименты с Kubernetes. Вас могло озадачить поведение Ingress. Вы скачали кучу YAML-файлов с примерами настройки, среди которых были Deployment и Ingress, и применили их к своему кластеру. После этого вы могли заметить, что Pod запустился без проблем, но обратиться к нему у вас не получалось. Ресурс Ingress фактически ничего не делал, и вы, наверное, спрашивали себя, что происходит?

Ingress — это один из тех API-интерфейсов, реализация которых делегируется создателям платформы. Иначе говоря, Kubernetes предоставляет интерфейс Ingress и ожидает, что он будет реализован другом компонентом. Этот компонент обычно называют *контроллером Ingress*.

Контроллер Ingress — это компонент платформы, который выполняется внутри кластера. Он отвечает за отслеживание событий Ingress API и их обработку в соответствии с конфигурацией, определенной в ресурсах Ingress. В большинстве реализаций контроллер Ingress применяется в сочетании с обратным прокси-сервером, таким как NGINX или Envoy. Такая двухкомпонентная архитектура сравнима с другими программно-определяемыми сетями в том смысле, что контроллер играет роль *плоскости управления*, а прокси-сервер выступает компонентом *плоскости данных*. Это продемонстрировано на рис. 6.10.

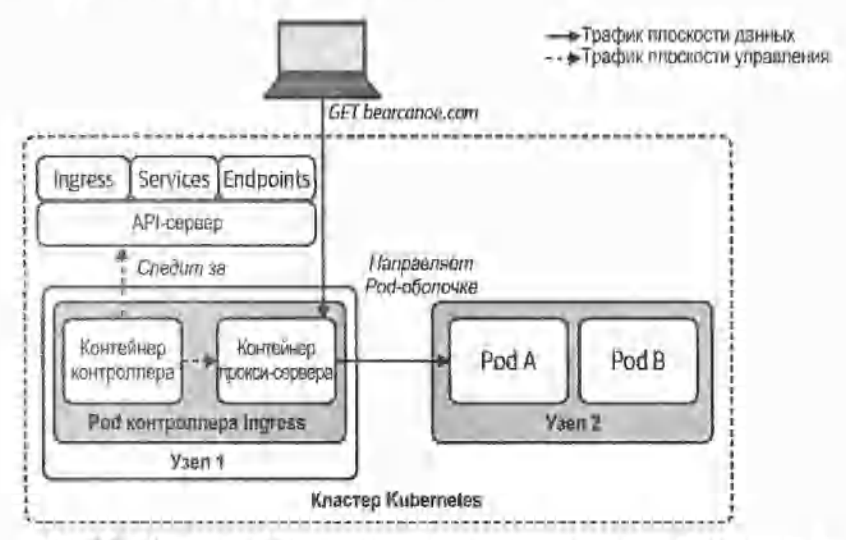


Рис. 6.10 Контроллер Ingress следит за различными ресурсами API-сервера и конфигурирует прокси-сервер соответствующим образом. Прокси-сервер обрабатывает входящий трафик и направляет его к Pod'ам в соответствии с конфигурацией Ingress

Плоскость управления контроллера Ingress подключается к API-интерфейсу Kubernetes и начинает следить за различными ресурсами, такими как Ingress, Services, Endpoints и др. Каждый раз, когда один из этих ресурсов меняется, контроллер получает уведомление и конфигурирует плоскость данных, действуя в соответствии с желаемым состоянием, объявленным в API-интерфейсе Kubernetes.

Плоскость данных занимается маршрутизацией и балансировкой сетевого трафика. Как уже упоминалось ранее, ее реализация делегируется готовому прокси-серверу.

Поскольку в основе API-интерфейса Ingress лежит абстрактный компонент Service, контроллеры Ingress могут послать трафик как через Сервисы, так и напрямую Pod'ам. Большинство выбирают второй вариант. Они используют ресурс `Service`, на который ссылается Ingress, только чтобы убедиться в его существовании. Что касается маршрутизации, то большинство контроллеров направляют трафик по IP-адресам Pod'ов, перечисленным в соответствующем объекте `Endpoints`. Это происходит в обход слоя Сервисов, что снижает латентность и расширяет выбор стратегий балансировки нагрузки.

Методы маршрутизации входящего трафика

Замечательная особенность Ingress состоит в том, что любое приложение имеет возможность настраивать маршрутизацию в соответствии со своими нуждами. У каждого приложения обычно есть свои требования к обработке входящего трафика. Некоторым из них может потребоваться прокси-сервер завершения TLS на граничном узле. А другие могут работать с TLS самостоятельно или вообще не поддерживать этот протокол (надеемся, это не ваш случай).

В этом разделе будут рассмотрены распространенные методы обработки входящего трафика, с которыми мы сталкивались. В итоге вы получите общее представление о том, чем объект Ingress может быть полезен вашим разработчикам и какую роль он может сыграть в вашей платформе.

Проксирование HTTP-трафика

Проксирование HTTP-трафика — это самый распространенный способ использования Ingress. Данный метод заключается в предоставлении доступа к одному или нескольким HTTP-сервисам и маршрутизации трафика в соответствии со свойствами HTTP-запросов. Мы уже обсуждали маршрутизацию на основе заголовка `Host`, в тех же целях можно задействовать и другие свойства, такие как URL-путь, метод запроса, заголовки запроса и прочие, в зависимости от контроллера Ingress.

Следующий ресурс `Ingress` открывает доступ к Сервису `app1` по адресу **app1.example.com**. Любой входящий запрос с соответствующим HTTP-заголовком `Host` направляется к Pod'у `app1` (листинг 6.11).

Листинг 6.11

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app1
spec:
  rules:
  - host: app1.example.com
```

```
    http:
      paths:
      - path: /
        backend:
          serviceName: app1
          servicePort: 80
```

Результатом применения этой конфигурации является плоскость данных, устроенная так, как показано на рис. 6.11.

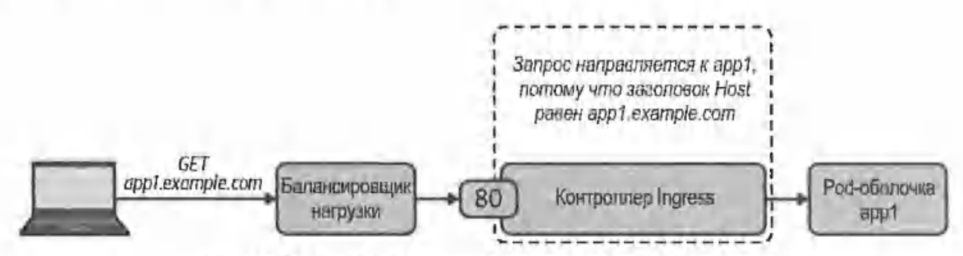


Рис. 6.11. Путь HTTP-запроса от клиента к Pod'у назначения, пролегающий через контроллер Ingress

Проксирование HTTP-трафика с использованием TLS

Поддержка шифрования TLS — это одно из основных требований к контроллерам Ingress. С точки зрения маршрутизации, входящий трафик обрабатывается так же, как и в случае с обычным протоколом HTTP. Но при этом клиенты взаимодействуют с контроллером Ingress по безопасному TLS-соединению, вместо того чтобы обмениваться обычным текстом.

В листинге 6.11 показан ресурс `Ingress`, который открывает доступ к `app1` по TLS. Контроллер получает сертификат TLS и ключ из объекта `Secret`, указанного в определении.

Листинг 6.11

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app1
spec:
  tls:
  - hosts:
      - app1.example.com
    secretName: app1-tls-cert
rules:
- host: app1.example.com
  http:
    paths:
    - path: /
```

```
    backend:
      serviceName: app1
      servicePort: 443
```

Контроллеры Ingress поддерживают разные конфигурации для соединений с внутренним сервисом. Между контроллером и внешним клиентом устанавливается безопасное соединение (TLS), а для взаимодействия с внутренним приложением это делать необязательно. Безопасность соединения между контроллером и внутренним компонентом зависит от того, принимает ли приложение трафик по TLS. Большинство контроллеров Ingress по умолчанию завершают сеанс TLS и направляют запросы по незашифрованному соединению, как показано на рис. 6.12.



Рис. 6.12. Контроллер Ingress обрабатывает HTTPS-запросы, завершая сеанс TLS и направляя запрос к внутреннему Pod'у по незашифрованному соединению

Если вам необходимо безопасное соединение с внутренним компонентом, контроллер Ingress завершает сеанс TLS на граничном узле и устанавливает новое TLS-соединение (как проиллюстрировано на рис. 6.13). Это не подходит для определенных приложений, например, для тех, которым нужно выполнять квитирование TLS со своими клиентами. Разумная альтернатива в таких ситуациях — сквозной режим TLS, который мы обсудим позже.

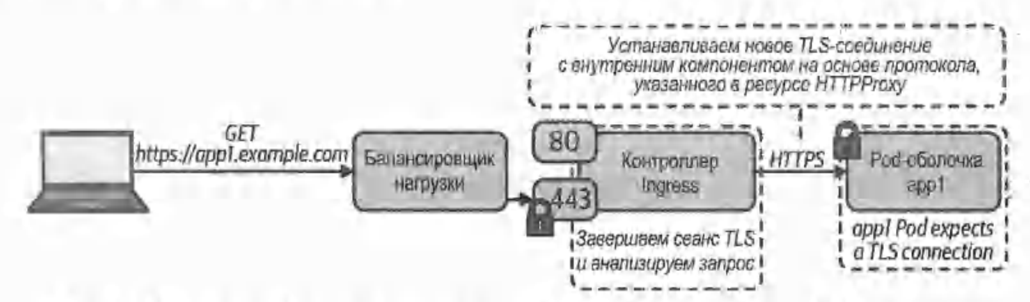


Рис. 6.13. Контроллер Ingress завершает сеанс TLS и устанавливает новое TLS-соединение с внутренним Pod'ом при обработке HTTPS-запросов

Проксирование на сетевом и транспортном уровнях

Несмотря на то, что API-интерфейс Ingress предназначен в первую очередь для проксирования пакетов прикладного уровня (HTTP-трафика, L7), контроллеры

Ingress могут работать с трафиком сетевого и транспортного уровней (TCP/UDP, L3/L4). Это может быть полезно, если вам нужно открыть доступ к приложению, которое не поддерживает HTTP. Об этом нужно помнить при выборе контроллера Ingress, так как не все они поддерживают проксирование трафика на уровнях L3/L4.

Основная трудность, возникающая при проксировании сервисов TCP и UDP, состоит в том, что контроллеры Ingress прослушивают ограниченное количество портов (обычно 80 и 443). Как можно себе представить, предоставление доступа к сервисам TCP или UDP на одном и том же порту требует какой-то стратегии распознавания трафика. Контроллеры Ingress решают это проблему по-разному. Некоторые из них, такие как Contour, поддерживают проксирование только TCP-соединений, зашифрованных с помощью TLS и расширения SNI (Server Name Indication). Это связано с тем, что контроллеру Contour нужно знать, куда направляется трафик. При использовании SNI целевое доменное имя доступно в незашифрованном виде в сообщении ClientHello, которое передается в процессе квитирования TLS. Поскольку TLS и SNI полагаются на TCP, Contour не поддерживает проксирование UDP.

Листинг 6.12 иллюстрирует пример пользовательского ресурса `HTTPProxy`, который поддерживается контроллером Contour. Проксирование трафика на уровнях L3/L4 — это один из тех случаев, когда пользовательский ресурс превосходит по возможностям API-интерфейс Ingress.

Листинг 6.12

```
apiVersion: projectcontour.io/v1
kind: HTTPProxy
metadata:
  name: tcp-proxy
spec:
  virtualhost:
    fqdn: tcp.bearcanoe.com
    tls:
      secretName: secret
  tcpproxy:
    services:
    - name: tcp-app
      port: 8080
```

В этой конфигурации Contour считывает имя сервера в расширении SNI и проксирует трафик к внутреннему TCP-сервису. Данная возможность проиллюстрирована на рис. 6.14.

Некоторые контроллеры Ingress предоставляют конфигурационные параметры, с помощью которых можно сделать так, чтобы прокси-сервер выделил дополнительные порты для проксирования трафика на уровнях L3/L4. Затем эти порты можно привязать к определенным сервисам, которые работают в кластере. Именно этот подход к проксированию трафика на сетевом и транспортном уровнях применяет контроллер NGINX Ingress, развитием которого занимается сообщество Kubernetes.

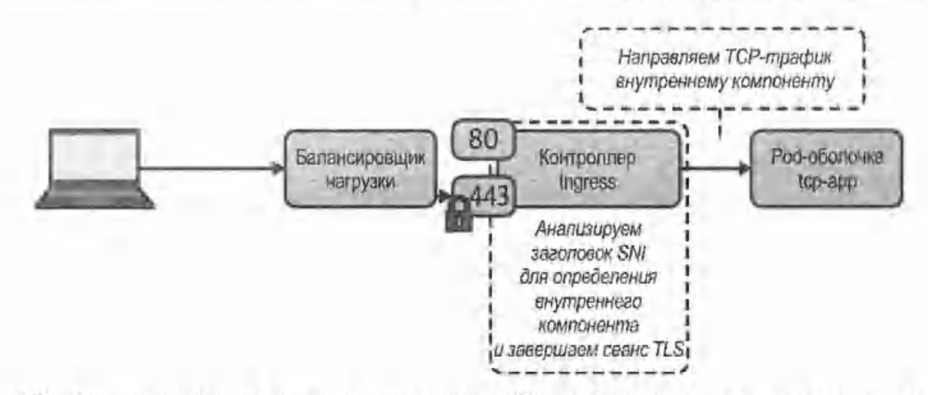


Рис. 6.14 Контроллер Ingress анализирует заголовок SNI, чтобы определить внутренний компонент, завершает сеанс TLS и направляет TCP-трафик к Pod'у

Проксирование трафика на уровнях L3/L4 зачастую применяется в рамках сквозного режима TLS. Этот режим используется приложениями, у которых есть конечная точка, доступная по TLS, и которым нужно выполнять квитирование TLS непосредственно с клиентом. Как уже обсуждалось ранее в разделе "Проксирование HTTP-трафика с использованием TLS", контроллер Ingress обычно разрывает TLS-соединения с клиентами. Это необходимо для того, чтобы контроллер Ingress мог проанализировать HTTP-запрос, который в противном случае оставался бы зашифрованным. Однако при реализации сквозного режима TLS контроллер Ingress не завершает сеанс TLS, а проксирует зашифрованный трафик внутреннему Pod'у. Этот процесс изображен на рис. 6.15.

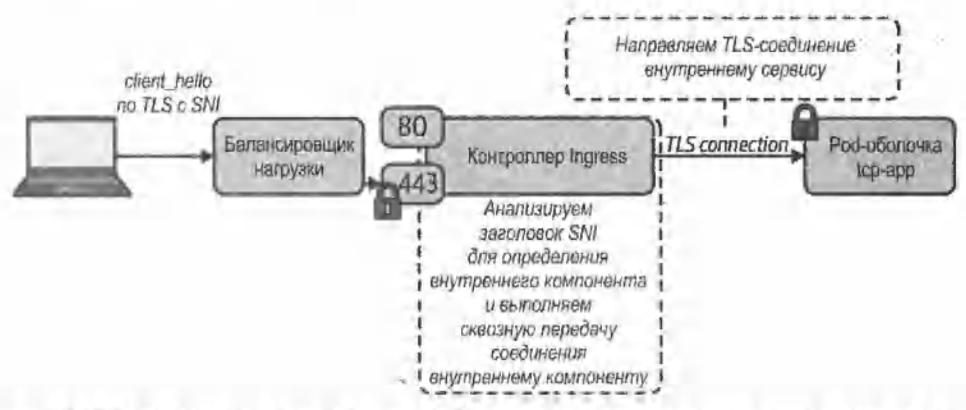


Рис. 6.15 Когда включен сквозной режим TLS, контроллер Ingress анализирует заголовок SNI, чтобы определить внутренний компонент, и направляет TLS-соединение соответствующим образом

Выбор контроллера Ingress

Вам на выбор доступно несколько контроллеров Ingress. Как показывает наш опыт, один из самых распространенных — NGINX Ingress. Но это не означает, что он лучше всего подходит для вашей платформы приложений. Есть и другие варианты, такие как Contour, HAProxy, Traefik и т. д. Придерживаясь духа этой книги, мы

не пытаемся навязывать вам свое мнение. Вместо этого мы хотим предоставить вам информацию, на основе которой вы сможете принять собственное решение. Мы также при необходимости обратим ваше внимание на существенные плюсы и минусы.

Вернемся немного назад и вспомним, что основное назначение контроллера Ingress состоит в управлении трафиком приложения. В связи с этим будет вполне естественно рассматривать приложения в качестве основного фактора при выборе контроллера Ingress. В частности выяснить, какие у вашего приложения требования, какие возможности ему нужны? Далее перечислены критерии, на основе которых вы можете оценивать контроллеры с точки зрения приложений:

- Предоставляют ли приложения конечные точки по HTTP? Нужно ли им заниматься квитированием TLS напрямую с клиентом? Возможно, им подойдет завершение сеанса TLS на граничном узле?
- Какие шифры SSL используют приложения?
- Нужна ли приложениям поддержка подобия сессий или липких сессий?
- Нужны ли приложениям дополнительные возможности маршрутизации запросов, основанные на заголовках, cookie, HTTP-методах и пр.?
- Требуются ли приложениям разные алгоритмы балансировки нагрузки, такие как циклический перебор, метод взвешенных наименьших запросов или случайный выбор?
- Нужна ли приложениям поддержка COSR (Cross-Origin Resource Sharing — совместное использование ресурсов между разными источниками)?
- Делегируют ли приложения вопросы аутентификации внешней системе? Некоторые контроллеры Ingress предоставляют возможности, на основе которых можно создать аутентификационный механизм, общий для всех приложений.
- Есть ли у вас приложения, которым нужно открывать доступ к конечным точкам по протоколам TCP или UDP?
- Нужна ли приложению возможность ограничивать входящий трафик?

Помимо требований, которые предъявляют ваши приложения, необходимо также учитывать опыт использования соответствующей технологии плоскости данных, которым обладает ваша организация. Если вы уже хорошо знакомы с определенным прокси-сервером, начинать обычно стоит именно с него. Вы уже хорошо разбираетесь в том, как он работает и, что более важно, вы знаете его ограничения и умеете его диагностировать.

Еще одним ключевым фактором, на который нужно обратить внимание, является техническая поддержка. Ingress — это неотъемлемый компонент вашей платформы. Он находится прямо между вашими клиентами и сервисами, к которым они хотят обращаться. Когда контроллер Ingress выходит из строя, у вас должна быть возможность обратиться за помощью.

Наконец, не забывайте, что вы можете применять в своей платформе сразу несколько контроллеров входящего трафика, используя классы Ingress. Такой шаг усложнит управление платформой, но в некоторых случаях это необходимо. Чем

больше у платформы пользователей и приложений, тем более высокие требования они будут предъявлять к возможностям вашего слоя Ingress. Вполне вероятно, что один-единственный Ingress будет не в состоянии удовлетворить всем предъявленным требованиям.

Вопросы, связанные с развертыванием контроллера Ingress

Какой бы контроллер Ingress вы ни выбрали, существует ряд факторов, которые необходимо учитывать при его развертывании и использовании. Некоторые из них могут также отразиться на приложениях, которые выполняются на вашей платформе.

Узлы, выделенные для Ingress

Как показывает наш опыт, выделение (или резервирование) набора узлов для выполнения контроллера Ingress, которые в результате будут служить "границей" кластера — это один из очень многих методов. Этот способ развертывания проиллюстрирован на рис. 6.16. На первый взгляд использование выделенных узлов для обработки входящего трафика может показаться расточительным. Но, согласно нашей философии, если вы можете себе позволить отдельные узлы для плоскости управления, у вас, вероятно, есть возможность выделить узлы для слоя, который играет важнейшую роль для всех приложений в кластере. Наличие выделенного пула узлов для Ingress имеет существенные преимущества.

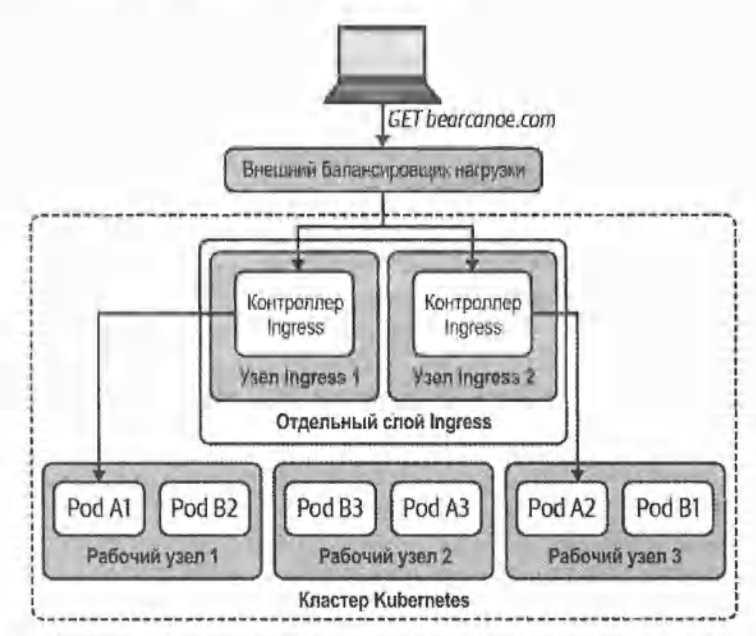


Рис. 6.16. Выделенные узлы Ingress зарезервированы для контроллера Ingress. Узлы, обрабатывающие входящий трафик, служат "границей" кластера или слоем Ingress

Основное преимущество этого подхода состоит в изоляции ресурсов. Несмотря на то, что Kubernetes поддерживает запросы и ограничения ресурсов, наш опыт показывает, что у разработчиков платформ могут возникать трудности при подборе этих параметров. Это особенно характерно для команд, которые только начинают переходить на Kubernetes и не ориентируются в деталях реализации, лежащих в основе механизма управления ресурсами (например, Completely Fair Scheduler, cgroups). Более того, на момент написания этих строк Kubernetes не поддерживает изоляцию ресурсов для сетевого ввода/вывода и файловых дескрипторов, что делает обеспечение их справедливого разделения затруднительным.

Еще одна причина размещения контроллеров Ingress на выделенных узлах — соблюдение нормативно-правовых требований. Мы не раз сталкивались с тем, что правила брандмауэров и другие средства безопасности, применяемые в организации, несовместимы с контроллерами Ingress. Обработка входящего трафика с помощью выделенных узлов в таких окружениях является полезным, так как сделать исключение для какой-то части кластера обычно легче, чем для всей системы целиком.

Наконец, ограничение числа узлов, предназначенных для выполнения контроллера Ingress, может пригодиться в системах с физической или локально размещенной инфраструктурой. В таких конфигурациях слой Ingress размещается за аппаратным балансировщиком нагрузки. В большинстве случаев это традиционные балансировщики без API-интерфейса, и, чтобы они могли направлять трафик определенному набору внутренних компонентов, их нужно настраивать вручную. Наличие небольшого числа узлов для обработки входящего трафика упрощает процесс настройки и администрирования этих внешних балансировщиков нагрузки.

В целом выделение узлов для Ingress может положительно сказаться на производительности, соблюдении нормативно-правовых требований и управлении внешними балансировщиками нагрузки. Лучше всего этот подход реализовывать путем назначения соответствующим узлам меток и ограничений (англ. taints). Затем контроллер Ingress развертывается в виде объекта `DaemonSet`, который допускает указанные ограничения и содержит селектор, охватывающий заданные узлы. При таком подходе необходимо быть готовым к тому, что узлы, обрабатывающие входящий трафик, могут выходить из строя, так как контроллер Ingress не сможет работать ни на каких других узлах. В идеале вместо неисправных узлов должны развертываться новые, способные продолжить обработку входящего трафика.

Привязка к сети хоста

Чтобы оптимизировать путь, который проходит входящий трафик, контроллер Ingress можно привязать к сети хоста. В результате входящие запросы будут приходить непосредственно в контроллер Ingress, минуя слой Сервисов. При использовании сети хоста убедитесь в том, что в качестве политики DNS контроллера Ingress указана `ClusterFirstWithHostNet`. В следующем фрагменте кода показан шаблон Pod'а с конфигурацией сети хоста и политики DNS:

```
spec:
  containers:
  - image: nginx
```

```
    name: nginx
  dnsPolicy: ClusterFirstWithHostNet
  hostNetwork: true
```

Использование контроллера Ingress напрямую в сети хоста может повысить производительность, но следует помнить, что при этом исчезают сетевые пространства имен, которые отделяют этот контроллер от узла. Иными словами, контроллер Ingress получает полный доступ ко всем сетевым интерфейсам и сервисам, доступным на хосте. Это ухудшает безопасность. В частности, злоумышленникам становится легче эксплуатировать уязвимости прокси-сервера в плоскости данных. Кроме того, операция присоединения к сети хоста является привилегированной, поэтому контроллер Ingress должен обладать повышенными привилегиями или исключениями для выполнения в качестве привилегированного приложения.

Но, как мы обнаружили, даже несмотря на эти недостатки, привязка к сети хоста себя оправдывает и, как правило, является лучшим способом предоставления доступа к контроллерам Ingress платформы. Входящий трафик поступает непосредственно в контроллер, минуя стек Сервисов (прохождение через который может быть неоптимальным вариантом, о чем мы уже упоминали в разделе "Сервисы Kubernetes" данной главы).

Контроллеры Ingress и политика маршрутизации внешнего трафика

Если Сервис Kubernetes не сконфигурирован как следует, его использование для предоставления доступа к контроллеру Ingress повлияет на производительность плоскости данных Ingress.

Как вы помните из раздела "Сервисы Kubernetes", политика маршрутизации внешнего трафика, принадлежащая Сервису, определяет то, как будет обрабатываться трафик, поступающий из-за пределов кластера. Если доступ к вашему контроллеру Ingress открыт с помощью Сервиса NodePort или LoadBalancer, убедитесь в том, что поле `externalTrafficPolicy` равно `Local`.

Политика `Local` позволяет избежать лишних сетевых переходов, так как внешний трафик поступает в локальный контроллер Ingress, не направляясь к другому узлу. Более того, политика `Local` не использует SNAT, значит приложения, обрабатывающие запросы, видят IP-адрес клиента.

Распределите контроллеры Ingress между разными доменами отказа

Чтобы обеспечить высокую доступность ваших контроллеров Ingress, распределите их по разным доменам отказа согласно правилам раздельного существования.

DNS-сервер и его роль в обработке входящего трафика

Как уже обсуждалось ранее в этой главе, приложения, выполняемые на платформе, имеют общую плоскость данных для входящего трафика, т. е. общую точку входа в

сеть платформы. Основная обязанность контроллера Ingress при поступлении запросов — классификация трафика и его маршрутизация в соответствии с параметрами Ingress.

Один из главных способов определения того, куда должен быть направлен запрос, состоит в анализе целевого доменного имени (заголовка `Host` в случае с HTTP или SNI в случае с TCP), что превращает DNS-сервер в ключевой компонент вашей реализации Ingress. Мы обсудим две основные модели совместного использования Ingress и DNS.

DNS-запись с подстановочными символами

Один из самых успешных подходов, который мы постоянно применяем, заключается в назначении среде доменного имени и разделении его на поддомены, предназначенные для разных приложений. Мы иногда называем это "маршрутизацией на основе поддоменов". Реализация этой стратегии требует создания DNS-записи с подстановочными символами (например, `*.bearcanoe.com`), которая ссылается на слой Ingress. Обычно это балансировщик нагрузки, размещенный перед контроллерами Ingress.

Преимущества DNS-записи с подстановочными символами для контроллеров Ingress:

- Приложениям в рамках своего домена может быть доступен любой путь, в том числе и корневой (/). Разработчикам не нужно тратить свое время на то, чтобы сделать свои приложения совместимыми с относительными путями. Некоторые приложения рассчитаны на размещение по корневому пути и в противном случае не работают.
- Реализация DNS относительно простая и понятная. Интеграции между Kubernetes и провайдером DNS не требуется.
- Одна-единственная DNS-запись с подстановочными символами решает все проблемы с распространением информации о доменных именах, возникающие при использовании отдельного поддомена для каждого приложения.

Интеграция Kubernetes и DNS

Альтернатива DNS-записи с подстановочными символами — интеграция платформы с DNS-провайдером. Сообщество Kubernetes занимается развитием контроллера под названием external-dns (**https://github.com/kubernetes-sigs/external-dns**), который предоставляет такого рода интеграцию. Если ваш DNS-провайдер поддерживается, можете воспользоваться этим контроллером для автоматизации процесса создания доменных имен.

Как можно было бы ожидать от контроллера Kubernetes, external-dns непрерывно согласовывает DNS-записи вашего DNS-провайдера и конфигурацию, определенную в ресурсе Ingress. Иными словами, external-dns создает, обновляет и удаляет DNS-записи в соответствии с изменениями, происходящими в API-интерфейсе Ingress. Для конфигурации DNS-записей этому контроллеру нужно два элемента

информации, каждый из которых является частью ресурса Ingress: желаемое доменное имя, указанное в спецификации Ingress, и IP-адрес назначения, доступный в поле `status` данного ресурса.

Интеграция платформы и DNS-провайдера может пригодиться в случае, если вам нужно поддерживать несколько доменных имен. Контроллер автоматически создает DNS-записи по мере необходимости. Но при этом необходимо учитывать следующие недостатки данного подхода:

- Вам потребуется развернуть в своем кластере дополнительный компонент (external-dns). Его нужно администрировать, поддерживать, отслеживать, версионировать и обновлять, что сделает ваши развертывания более сложными.
- Если у external-dns нет поддержки вашего DNS-провайдера, вам придется разработать собственный контроллер. Его создание и поддержка потребует инженерных ресурсов, которые можно было бы направить на что-то более полезное. В таких ситуациях лучше всего просто воспользоваться DNS-записью с подстановочными символами.

Управление сертификатами TLS

Для обслуживания приложений по TLS контроллерам Ingress нужны сертификаты и прилагающиеся к ним закрытые ключи. В зависимости от выбранной вами стратегии, управление сертификатами может оказаться громоздким процессом. Если ваш кластер имеет всего одно доменное имя и реализует маршрутизацию на основе поддоменов, вы можете использовать один групповой сертификат TLS. Но в некоторых случаях кластеры охватывают разные домены, что затрудняет управление сертификатами. Более того, ваш отдел информационной безопасности может относиться к групповым сертификатам с опаской. Как бы то ни было, сообщество Kubernetes направило свои усилия на развитие дополнения с метким названием cert-manager (**https://cert-manager.io**), которое упрощает создание и администрирование сертификатов.

Cert-manager — это контроллер, который работает в вашем кластере. Он устанавливает ряд пользовательских ресурсов, которые делают возможным декларативное управление центрами сертификации (ЦС) и сертификатами посредством API-интерфейса Kubernetes. Что еще более важно, он поддерживает разные организации, которые занимаются выдачей сертификатов, включая ЦС на основе ACME, HashiCorp Vault, Venafi и т. д. Он также предоставляет возможность расширения, чтобы вы при необходимости могли реализовать собственные ЦС.

Cert-manager содержит два пользовательских ресурса для получения сертификатов. Ресурс Issuer представляет ЦС, который подписывает сертификаты для определенного пространства имен Kubernetes. Если вы хотите получить сертификат, предназначенный для всех пространств имен, можете использовать ресурс ClusterIssuer.

Далее приведен пример определения `ClusterIssuer`, в котором используется закрытый ключ, хранящийся в объекте `Secret` с именем `platform-ca-key-pair`.

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: prod-ca-issuer
spec:
  ca:
    secretName: platform-ca-key-pair
```

Замечательная особенность контроллера cert-manager заключается в том, что он интегрируется с API-интерфейсом Ingress для автоматической выдачи сертификатов ресурсам Ingress. Например, если взять следующий объект Ingress, cert-manager автоматически создаст пару "сертификат – ключ", которая подходит для TLS (листинг 6.13).

Листинг 6.13

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: prod-ca-issuer ❶
  name: bearcanoe-com
spec:
  tls:
  - hosts:
    - bearcanoe.com
    secretName: bearcanoe-cert-key-pair ❷
  rules:
  - host: bearcanoe.com
    http:
      paths:
      - path: /
        backend:
          serviceName: nginx
          servicePort: 80
```

❶ Аннотация `cert-manager.io/cluster-issuer` заставляет cert-manager использовать `prod-ca-issuer` для выдачи сертификатов.

❷ Cert-manager хранит сертификаты и закрытые ключи в объекте `Secret` с именем `bearcanoe-cert-key-pair`.

Внутри cert-manager происходит процесс запрашивания сертификатов, который включает в себя генерацию закрытого ключа, создание запроса на подпись сертификата (англ. Certificate Signing Request или CSR) и подачу CSR на рассмотрение в центр сертификации. Как только ЦС выдаст сертификат, cert-manager сохранит его в объекте `bearcanoe-cert-key-pair`. Затем контроллер Ingress сможет взять этот сер-

тификат и направлять с его помощью трафик к приложениям по TLS. На рис. 6.17 этот процесс изображен более подробно.

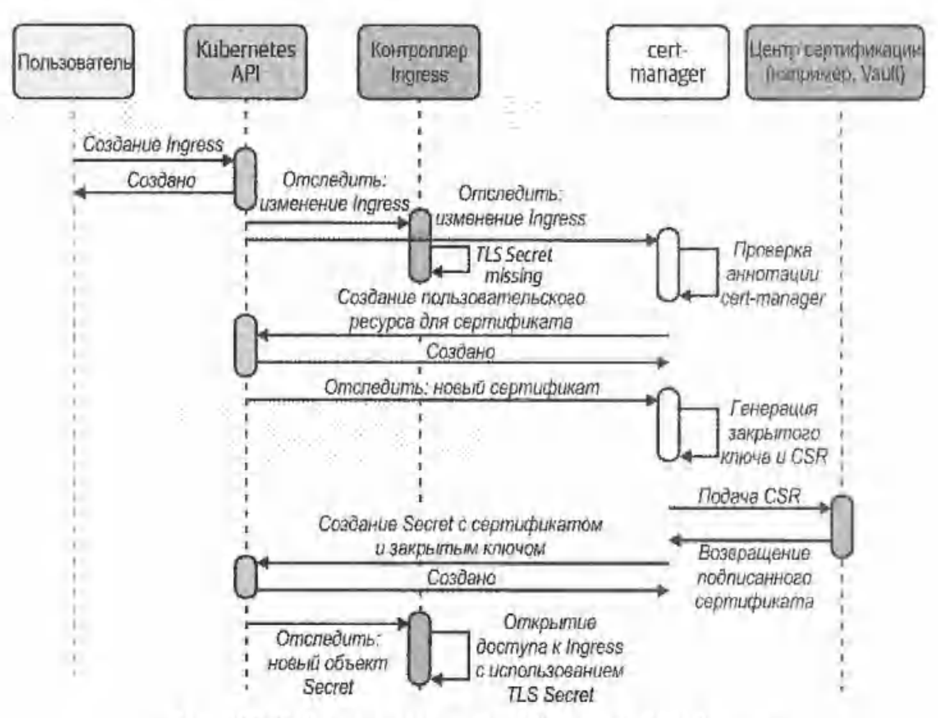


Рис. 6.17. Cert-manager следит за API-интерфейсом Ingress и при обнаружении ресурса Ingress с аннотацией `cert-manager.io/cluster-issuer` запрашивает сертификат у центра сертификации

Как видите, контроллер cert-manager упрощает управление сертификатами в Kubernetes. Он так или иначе задействован в большинстве платформ, которые нам встречались. Если вы применяете его в своей платформе, можете дополнить его внешней системой, такой как Vault или ЦС. Интеграция cert-manager с внешней системой вместо внутреннего ЦС на основе объектов Secret — более надежное и безопасное решение.

Mesh-сеть

По мере того, как организации продолжают внедрять у себя контейнеры и микросервисы, огромной популярностью начали пользоваться mesh-сети (англ. service mesh). Термин "mesh-сеть" относительно новый, чего нельзя сказать о самой идее, которая за ним стоит. Mesh-сети представляют собой очередную модификацию ранее существовавших веяний в сфере маршрутизации сервисов, балансировки нагрузки и телеметрии. До того, как контейнеры и Kubernetes получили широкое распространение, Интернет-гиганты решали трудности, связанные с микросервисами, за счет использования предшественников mesh-сетей. Например, разработчики Twitter создали библиотеку Finagle (**https://twitter.github.io/finagle**) на языке Scala,

которая встраивалась во все микросервисы компании. Она отвечала за балансировку нагрузки, реализацию шаблона проектирования "предохранитель", автоматическое повторение операций, телеметрию и т. д. Компания Netflix разработала Hystrix (**https://github.com/Netflix/Hystrix**), похожую библиотеку для приложений на Java.

Ситуация поменялась с приходом контейнеров и Kubernetes. Mesh-сети, в отличие от их предшественников, представляют собой не библиотеки для конкретных языков, а распределенные системы. Они состоят из плоскости управления, которая конфигурирует набор прокси-серверов, реализующих плоскость данных. Функции, относящиеся к маршрутизации, балансировке нагрузки, телеметрии и пр. встраиваются в прокси-сервер, а не в приложение. Переход на такую модель позволил распространить эти возможности на более широкий круг приложений, так как для участия в mesh-сети не нужно вносить никаких дополнительных изменений в код.

Mesh-сети имеют широкие возможности, которые можно разделить на три основные категории:

- *Маршрутизация и надежность*. Передовые механизмы маршрутизации и обеспечения доступности, такие как смещение и зеркалирование трафика, повторное выполнение неудавшихся операций и шаблон проектирования "предохранитель".
- *Безопасность*. Средства идентификации и управления доступом, обеспечивающие безопасное взаимодействие между сервисами, включая управление сертификатами и mTLS.
- *Наблюдаемость*. Автоматический сбор метрик и трассировка всех взаимодействий, происходящих в mesh-сети.

Оставшаяся часть этой главы посвящена более подробному рассмотрению mesh-сетей. Но сначала давайте вернемся к центральной теме этой книги и спросим себя, "нужна ли нам mesh-сеть"? Популярность mesh-сетей продиктована тем фактом, что некоторые организации считают их идеальным средством для реализации перечисленных ранее возможностей. Но, как показывает наш опыт, прежде чем внедрять mesh-сеть, стоит тщательно взвесить последствия этого шага.

Где (не) следует использовать mesh-сети

Mesh-сеть может принести огромную пользу платформе приложений и самим приложениям, которые работают поверх нее. Она имеет привлекательный набор возможностей, которые по достоинству оценят ваши разработчики. Но в то же время mesh-сеть создает множество сложностей, которые требуют к себе внимания.

Kubernetes — это сложная распределенная система. До сих пор в этой книге затрагивались некоторые составные элементы, необходимые для создания платформы приложений поверх Kubernetes, и впереди еще много глав. Но создание успешной платформы приложений на основе Kubernetes требует много усилий. Помните об этом, когда будете принимать решение об использовании mesh-сети. Работа над реализацией mesh-сети, когда вы только начинаете свой переход на Kubernetes, замедлит ваш прогресс, а возможно, даже заведет вас в тупик.

Мы знакомы с такими случаями не понаслышке. Мы сотрудничали с создателями платформ, которые были ослеплены блестящими возможностями mesh-сети. Конечно, это сделало бы их платформу более привлекательной для разработчиков и, следовательно, способствовало бы ее внедрению. Но всему свое время. Прежде чем задумываться о mesh-сетях, наберитесь опыта работы с системами в условиях эксплуатации.

Наверное, еще более важным является понимание ваших требований или проблем, которые вы пытаетесь решить. Пытаясь бежать впереди паровоза, вы не только снизите вероятность создания удачной платформы, но и впустую потратите инженерные ресурсы. В качестве наглядного примера подобной ошибки приведем организацию, которая взялась за mesh-сеть в то время, когда ее платформа, основанная на Kubernetes, была еще не готова к внедрению. "Мы хотим mesh-сеть, потому что нам нужны все те возможности, которые она предоставляет", — заявило руководство. Двенадцать месяцев спустя единственной задействованной функцией mesh-сети оказался механизм обработки входящего трафика. Никакого mTLS, никакой нетривиальной маршрутизации или трассировки. Только Ingress. Вместо реализации полноценной mesh-сети было бы куда экономней подготовить к эксплуатации отдельный контроллер Ingress. Иногда лучше сосредоточиться на создании продукта, удовлетворяющего минимальным требованиям, а уже затем заниматься расширением его возможностей.

К этому моменту у вас может возникнуть ощущение, что мы не видим места для mesh-сети в платформе приложений. Но все с точностью до наоборот. Mesh-сеть может решить огромное количество проблем, если таковые имеются, и принести большую пользу, если вы готовы воспользоваться ее преимуществами. В итоге мы пришли к тому, что для успешной реализации mesh-сети ее внедрение должно быть обоснованным и проводиться в подходящий момент.

Интерфейс mesh-сети

Kubernetes предоставляет интерфейсы для разнообразных подключаемых компонентов, включая CRI (Container Runtime Interface — интерфейс среды выполнения контейнеров), CNI (Container Networking Interface — интерфейс управления сетью контейнеров) и др. Как вы уже видели на страницах этой книги, именно данные интерфейсы делают платформу Kubernetes настолько расширяемой. Mesh-сети медленно, но уверенно становятся еще одним таким компонентом. В связи с этим сообщество объединило усилия для разработки интерфейса mesh-сетей (англ. Service Mesh Interface или SMI).

Подобно другим интерфейсам, которые мы уже обсуждали, SMI определяет взаимодействие между Kubernetes и mesh-сетями. Однако у SMI есть одна особенность, которая выделяет его на фоне других интерфейсов: он не является частью основного проекта Kubernetes. Вместо этого проект SMI описывает свой интерфейс с помощью пользовательских ресурсов. В состав SMI также входят библиотеки реализации, такие как SMI SDK для Go.

Со своим набором пользовательских ресурсов SMI охватывает три основные области, которые мы рассмотрели в предыдущем разделе. Traffic Split API занимается маршрутизацией и распределением трафика между сервисами. Этот интерфейс поддерживает разделение трафика на основе процентного соотношения, что делает возможными разные модели доставки кода, такие как сине-зеленые развертывания и A/B-тестирование. В листинге 6.14 приведен пример ресурса TrafficSplit, который выполняет канареечное развертывание веб-сервиса "flights".

Листинг 6.14

```
apiVersion: split.smi-spec.io/v1alpha3
kind: TrafficSplit
metadata:
  name: flights-canary
  namespace: bookings
spec:
  service: flights ❶
  backends: ❷
  - service: flights-v1
    weight: 70
  - service: flights-v2
    weight: 30
```

❶ Сервис верхнего уровня, к которому подключаются клиенты (т. е. `flights.bookings.cluster.svc.local`).

❷ Внутренние Сервисы, которые принимают трафик. Версия v1 получает 70 % трафика, а версии v2 достается все остальное.

API-интерфейсы Traffic Access Control и Traffic Specs совместно реализуют механизмы безопасности такие, как управление доступом. Traffic Access Control предоставляет CRD для определения того, какие взаимодействия сервисов допускаются в mesh-сети. С помощью этих CRD разработчики могут описать политику управления доступом, которая определяет, какие сервисы и при каких условиях (это, к примеру, может быть список HTTP-методов) могут взаимодействовать между собой. API-интерфейс Traffic Specs позволяет описывать трафик; ресурс `HTTPRouteGroup` предназначен для HTTP, а `TCPRoute` — для TCP. В сочетании с пользовательскими ресурсами Traffic Access Control это позволяет применять политику на прикладном уровне.

Например, ресурсы `HTTPRouteGroup` и `TrafficTarget` допускают все запросы от сервиса бронирования билетов к сервису платежей. Ресурс `HTTPRouteGroup` описывает трафик, а `TrafficTarget` указывает исходный и конечный сервисы (листинг 6.15).

Листинг 6.15

```
apiVersion: specs.smi-spec.io/v1alpha3
kind: HTTPRouteGroup
```

```
metadata:
  name: payment-processing
  namespace: payments
spec:
  matches:
  - name: everything ❶
    pathRegex: ".*"
    methods: ["*"]
---
apiVersion: access.smi-spec.io/v1alpha2
kind: TrafficTarget
metadata:
  name: allow-bookings
  namespace: payments
spec:
  destination: ❷
    kind: ServiceAccount
    name: payments
    namespace: payments
    port: 8080
  rules: ❸
  - kind: HTTPRouteGroup
    name: payment-processing
    matches:
    - everything
  sources: ❹
  - kind: ServiceAccount
    name: flights
    namespace: bookings
```

❶ В этом ресурсе `HTTPRouteGroup` разрешены все запросы.

❷ Сервис назначения. В данном случае это Pod'ы, использующие служебную учетную запись `payments` в пространстве имен `payments`.

❸ Ресурс `HTTPRouteGroups`, управляющий трафиком между исходным и конечным сервисами.

❹ Исходный сервис. В данном случае это Pod'ы, использующие служебную учетную запись `flights` в пространстве имен `bookings`.

Наконец, Traffic Metrics API предоставляет возможности телеметрии, доступные в mesh-сети. Этот API-интерфейс, в отличие от остальных, определяет вывод, а не механизмы для задания ввода. Это стандарт для открытия доступа к метрикам сервисов. Системы, которым нужны эти метрики, такие как средства мониторинга и автомасштабирования, информационные панели и прочие, могут потреблять их стандартным путем. В листинге 6.16 показан ресурс `TrafficMetrics`, который открывает доступ к метрикам для трафика между двумя Pod'ами.

Листинг 6.16

```
apiVersion: metrics.smi-spec.io/v1alpha1
kind: TrafficMetrics
resource:
  name: flights-19sk18sj11-a9od2
  namespace: bookings
  kind: Pod
edge:
  direction: to
  side: client
  resource:
    name: payments-ks8xoa999x-xkop0
    namespace: payments
    kind: Pod
timestamp: 2020-08-09T01:07:23Z
window: 30s
metrics:
- name: p99_response_latency
  unit: seconds
  value: 13m
- name: p90_response_latency
  unit: seconds
  value: 7m
- name: p50_response_latency
  unit: seconds
  value: 3m
- name: success_count
  value: 100
- name: failure_count
  value: 0
```

SMI — один из новейших интерфейсов в экосистеме Kubernetes. Он все еще развивается, но уже дает понять, в каком направлении мы, как сообщество, движемся. Как и другие интерфейсы Kubernetes, SMI позволяет разработчикам снабжать свою платформу mesh-сетью с переносимыми API, которые не зависят от конкретного провайдера, что только повышает ценность, гибкость и мощь Kubernetes.

Прокси-сервер плоскости данных

Плоскость данных mesh-сети представляет собой набор прокси-серверов, которые соединяют сервисы между собой. Один из самых популярных прокси-серверов в облачно-ориентированной экосистеме — Envoy (**https://www.envoyproxy.io**). Этот проект, изначально разработанный компанией Lyft, открыл свой исходный код в конце 2016 года (**https://oreil.ly/u5fCD**) и быстро получил широкое распространение в облачно-ориентированных системах.

Envoy используется в контроллерах Ingress (Contour [**https://projectcontour.io**]), API-шлюзах (Ambassador [**https://www.getambassador.io**], Gloo [**https://docs.solo.io/gloo/latest**]) и, как вы уже могли догадаться, в mesh-сетях (Istio [**https://istio.io**], OSM [**https://github.com/openservicemesh/osm**]).

Один из факторов, которые делают Envoy настолько хорошим составным элементом, заключается в поддержке динамической конфигурации посредством gRPC/REST API. Прокси-серверы с открытым исходным кодом, предшествовавшие Envoy, не были рассчитаны на такие динамические окружения, как Kubernetes. Они использовали статические конфигурационные файлы и требовали перезапуска для применения внесенных изменений. Для сравнения, Envoy предоставляет API-интерфейсы *x*DS (англ. Discovery Service — сервис обнаружения [чего-то]) для динамической конфигурации (рис. 6.18). Он также поддерживает быстрый перезапуск, что позволяет выполнять повторную инициализацию без разрыва каких-либо активных соединений.

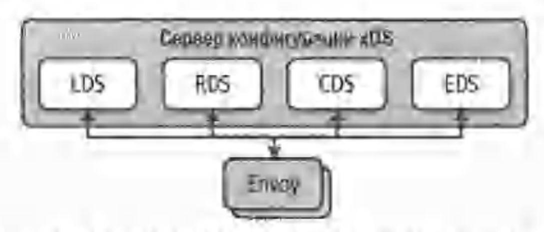


Рис. 6.18. Envoy поддерживает динамическую конфигурацию посредством API-интерфейсов *x*DS. Envoy подключается к серверу конфигурации и запрашивает свои параметры с помощью LDS, RDS, EDS, CDS и других интерфейсов *x*DS

Интерфейс *x*DS из состава Envoy — это набор API-интерфейсов, в который входят LDS (Listener Discovery Service — сервис обнаружения слушателей), CDS (Cluster Discovery Service — сервис обнаружения кластеров), EDS (Endpoints Discovery Service — сервис обнаружения конечных точек), RDS (Route Discovery Service — сервис обнаружения маршрутов) и др. *Сервер конфигурации* реализует эти API-интерфейсы и выступает источником динамической конфигурации для Envoy. Во время запуска Envoy обращается к этому серверу (обычно по gRPC) и подписывается на конфигурационные изменения. Когда в окружении что-то меняется, сервер конфигурации шлет Envoy поток изменений. Давайте рассмотрим API-интерфейсы *x*DS более подробно.

LDS API настраивает объекты Listener (Слушатели), которые служат точкой входа в прокси-сервер. Envoy может открыть несколько таких объектов, к которым смогут подключаться клиенты. Типичный пример — прослушивание портов 80 и 443 для HTTP- и HTTPS-трафика.

Каждый Слушатель имеет набор цепочек фильтров, которые определяют, как обрабатывается входящий трафик. Фильтр управления HTTP-соединениями использует RDS API для получения параметров маршрутизации, которые говорят Envoy о том, как маршрутизировать входящие HTTP-запросы. Он предоставляет подробности о виртуальных хостах и сопоставлении запросов (на основе путей, заголовков и т. д.).

Каждый маршрут в параметрах маршрутизации ссылается на *Кластер* — набор *Конечных точек*, принадлежащих одному сервису. Для обнаружения Кластеров и Конечных точек Envoy использует API-интерфейсы CDS и, соответственно, EDS. Интересно, что у EDS API нет объекта Endpoint как такового. Вместо этого Конечные точки назначаются кластерам с помощью объектов ClusterLoadAssignment.

Подробному рассмотрению API-интерфейсов *x*DS можно было бы посвятить отдельную книгу, но мы надеемся, что приведенный краткий обзор помог вам сориентироваться в принципах работы и возможностях Envoy. Если подытожить, то слушатели привязываются к портам и принимают соединения от клиентов. У слушателей есть цепочки фильтров, которые определяют, что делать с этими входящими соединениями. Например, HTTP-фильтр анализирует запросы и сопоставляет их с кластерами. У каждого кластера есть одна или несколько конечных точек, которые впоследствии принимают и обрабатывают трафик. На рис. 6.19 наглядно показаны эти концепции и отношения между ними.

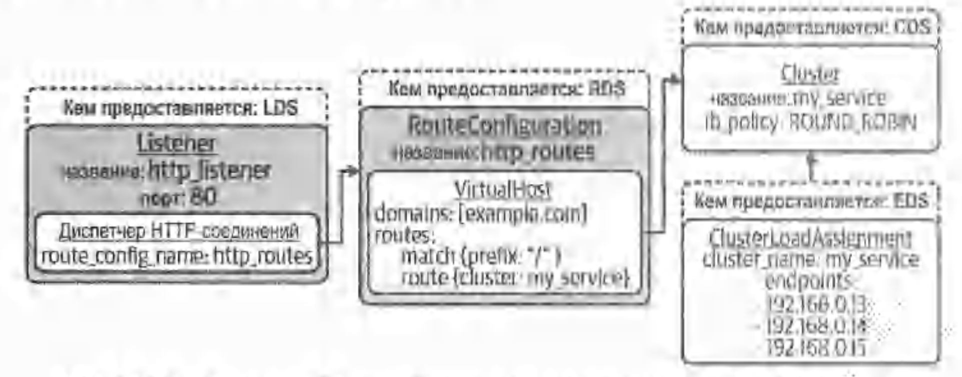


Рис. 6.19. Конфигурация Envoy со Слушателем, который привязывается к порту 80
У Слушателя есть фильтр управления HTTP-соединениями, который ссылается на параметры маршрутизации. Параметры маршрутизации сопоставляют запросы с префиксом / и направляют их кластеру `my_service` с тремя конечными точками

Mesh-сеть в Kubernetes

В предыдущем разделе мы обсудили то, как плоскость данных mesh-сети дает возможность сервисам взаимодействовать между собой. Мы также поговорили о прокси-сервере этой плоскости, Envoy, и о том, как он поддерживает динамическую конфигурацию посредством API-интерфейсов *x*DS. Для создания mesh-сети в Kubernetes нам нужна плоскость управления, которая будет конфигурировать ее плоскость данных в соответствии с происходящим внутри кластера. Плоскость управления должна уметь обращаться с Сервисами, Конечными точками, Pod'ами и т. д. Более того, она должна предоставлять доступ к пользовательским ресурсам, с помощью которых разработчики смогут конфигурировать mesh-сеть.

Одна из самых популярных реализаций mesh-сети для Kubernetes — Istio. Она предоставляет управляющую плоскость для mesh-сетей, основанных на Envoy. Эта плоскость имеет вид компонента под названием "istiod", который в свою очередь состоит из трех основных подкомпонентов: Pilot, Citadel и Galley. Pilot — это сер-

вер конфигурации Envoy. Он реализует API-интерфейсы *x*DS и направляет поток конфигурационных параметров прокси-серверам Envoy, которые выполняются вместе с приложениями. Компонент Citadel отвечает за управление сертификатами внутри mesh-сети. Он выдает сертификаты, которые требуются для идентификации сервисов и mTLS. Наконец, Galley взаимодействует с внешними системами, такими как Kubernetes, для получения конфигурации. Он абстрагируется от имеющейся платформы и передает конфигурацию другим компонентам istiod. На рис. 6.20 показано взаимодействие между компонентами плоскости управления Istio.

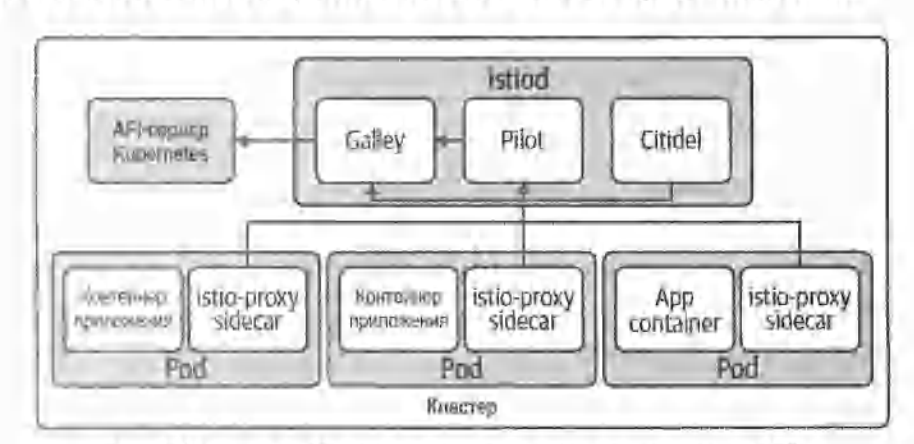


Рис. 6.20 Взаимодействие компонентов в плоскости управления Istio

Помимо конфигурации плоскости данных mesh-сети Istio обладает и другими возможностями. Прежде всего, у Istio есть веб-хук допуска с возможностью записи, который внедряет прокси-серверы Envoy в Pod. У каждого Pod'а, участвующего в mesh-сети, есть прокси-сервер Envoy, который обрабатывает все входящие и исходящие соединения. Этот веб-хук облегчает разработчикам использование платформы, так как им не нужно вручную добавлять прокси-серверы в манифесты развертывания всех своих приложений. Платформа делает это автоматически, позволяя явно указать те Pod'ы, в которые нужно (или не нужно) внедряться. Тем не менее, сам факт внедрения прокси-сервера в приложение вовсе не означает, что оно сразу же начнет отправлять свой трафик через Envoy. Поэтому Istio задействует контейнер инициализации для добавления правил iptables, которые перехватывают сетевой трафик Pod'а и направляют его к Envoy. В следующем (сокращенном) фрагменте кода (листинг 6.17) показана конфигурация контейнера инициализации Istio.

Листинг 6.17

```
...
initContainers:
- args:
  - istio-iptables
  - --envoy-port ➊
  - "15001"
```

```
  - --inbound-capture-port ❷
  - "15006"
  - --proxy-uid
  - "1337"
  - --istio-inbound-interception-mode
  - REDIRECT
  - --istio-service-cidr ❸
  - '*'
  - --istio-inbound-ports ❹
  - '*'
  - --istio-local-exclude-ports
  - 15090,15021,15020
  image: docker.io/istio/proxyv2:1.6.7
  imagePullPolicy: Always
  name: istio-init
...
```

❶ Istio создает правило iptables, которое берет весь исходящий трафик и шлет его прокси-серверу Envoy на этот порт.

❷ Istio создает правило iptables, которое берет весь входящий трафик и шлет его прокси-серверу Envoy на этот порт.

❸ Список диапазонов IP-адресов, которые будут перенаправлены к Envoy. В данном случае мы перенаправляем все диапазоны.

❹ Список портов, которые будут перенаправлены к Envoy. В данном случае мы перенаправляем все порты.

Итак, мы обсудили архитектуру Istio. Теперь давайте рассмотрим некоторые востребованные возможности mesh-сетей. Одно из самых распространенных требований, с которыми мы сталкиваемся в этой области, — аутентификация сервисов и шифрование трафика, которым они обмениваются. За эту возможность отвечают API-интерфейсы Traffic Access Control из SMI. Istio и большинство других реализаций mesh-сети используют mTLS. В Istio режим mTLS включен по умолчанию для всех сервисов, которые входят в mesh-сеть. Приложение шлет незашифрованный трафик размещенному рядом прокси-серверу, а тот повышает уровень безопасности соединения до mTLS и передает его прокси-серверу на другом конце. По умолчанию трафик по-прежнему может приходить без шифрования TLS, но только если он был отправлен сервисом, не входящим в mesh-сеть. На случай, если вы хотите принудительно включить mTLS для всех взаимодействий, Istio поддерживает режим `STRICT`, который конфигурирует все сервисы в mesh-сети так, чтобы они принимали только запросы, зашифрованные с помощью TLS. Например, следующая конфигурация позволяет принудительно включить строгий режим mTLS на уровне кластера в пространстве имен `istio-system`:

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default"
  namespace: "istio-system"
```

```
spec:
  mtls:
    mode: STRICT
```

Еще одна ключевая обязанность, которую берет на себя mesh-сеть, — управление трафиком. Для этого в SMI предусмотрен интерфейс Traffic Split API, хотя аналогичные возможности Istio более развитые. Помимо разделения и смещения трафика Istio поддерживает внесение ошибок, шаблон проектирования "предохранитель", зеркалирование и др. Для конфигурации смещения трафика Istio использует два отдельных пользовательских ресурса: `VirtualService` и `DestinationRule`.

- `VirtualService` — создает сервисы в mesh-сети и определяет, как к ним направляется трафик. В нем указывается сетевое имя сервиса и правила для управления местом назначения запросов. Например, `VirtualService` может направить 90 % трафика в одно место, а остальные 10 % — в другое. Проанализировав правила и выбрав место назначения, `VirtualService` передает трафик определенному подмножеству объектов `DestinationRule`.
- `DestinationRule` — содержит список "реальных" внутренних компонентов, доступных для заданного Сервиса. Каждый компонент находится в отдельном подмножестве, а у каждого подмножества могут быть свои параметры маршрутизации, такие как политика балансировки нагрузки, режим mTLS и др.

Рассмотрим в качестве примера ситуацию, когда нам нужно медленно выкатить новую версию сервиса (v2). Для этого можно воспользоваться ресурсами `DestinationRule` и `VirtualService`. Первый создает два подмножества сервисов: v1 и v2, а второй на эти подмножества ссылается. `VirtualService` шлет 90 % трафика версии v1 и 10 % версии v2 (листинг 6.18).

Листинг 6.18

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: flights
spec:
  host: flights
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: flights
```

```
spec:
  hosts:
  - flights
  http:
  - route:
    - destination:
        host: flights
        subset: v1
      weight: 90
    - destination:
        host: flights
        subset: v2
      weight: 10
```

Еще одна востребованная возможность — наблюдаемость сервисов. Поскольку между всеми сервисами в mesh-сети находится прокси-сервер, процесс извлечения их метрик не составляет труда. Чтобы получить эти метрики, разработчикам не нужно заниматься инструментированием своих приложений. Соответствующая информация предоставляется в формате Prometheus, что делает ее доступной для широкого спектра систем мониторинга. В листинге 6.19 приведен пример метрики, захваченной "прицепным" прокси-сервером (некоторые метки были убраны для краткости); она говорит о том, что сервис бронирования билетов сделал 7 183 успешных запроса к сервису обработки платежей.

Листинг 6.19

```
istio_requests_total{
  connection_security_policy="mutual_tls",
  destination_service_name="payments",
  destination_service_namespace="payments",
  destination_version="v1",
  request_protocol="http",
  ...
  response_code="200",
  source_app="bookings",
  source_version="v1",
  source_workload="bookings-v1",
  source_workload_namespace="flights"
} 7183
```

В целом проект Istio поддерживает все возможности, предусмотренные в SMI, за исключением API-интерфейса SMI, который на момент выхода версии v1.6 еще не реализован. Сообщество SMI занимается развитием адаптера (**https://github.com/servicemeshinterface/smi-adapter-istio**), с помощью которого можно сделать так, чтобы API-интерфейсы SMI работали с Istio. Мы обсуждаем этот проект, так как именно эта mesh-сеть встречается нам чаще всего в реальных условиях. Однако в экосистеме Kubernetes существуют и другие mesh-сети, такие как Linkerd, Consul

Connect, Maesh и прочие. Отличительная черта всех этих реализаций — архитектура плоскости данных, о которой речь пойдет дальше.

Архитектура плоскости данных

Mesh-сеть — это шина, по которой сервисы могут взаимодействовать между собой. Для доступа к ней сервисы используют прокси-сервер. Что касается плоскости данных, то mesh-сети реализуются по одной из двух архитектурных моделей: "прицепной" прокси-сервер (sidecar-прокси) или узловой прокси-сервер (node прокси).

Sidecar-прокси

Sidecar-прокси наиболее распространен среди этих двух архитектурных моделей. Как уже обсуждалось в предыдущем разделе, в Istio он применяется для реализации плоскости данных с помощью прокси-серверов Envoy. Такой же подход можно наблюдать в Linkerd. В сущности, mesh-сети, спроектированные по этой модели, развертывают прокси-сервер внутри Pod'а приложения, рядом с сервисом. Этот sidecar-прокси перехватывает весь трафик, который входит и выходит из сервиса, как показано на рис. 6.21.

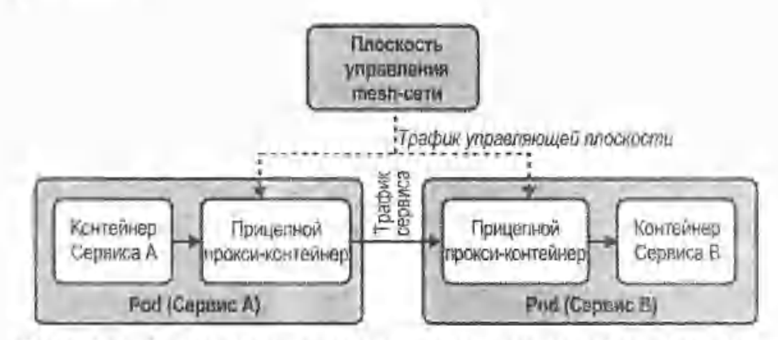


Рис. 6.21. У Pod'ов, входящих в mesh-сеть, есть прицепной прокси-сервер, который перехватывает их сетевой трафик

Если сравнивать с подходом, в котором прокси-сервер назначается целым узлам, архитектура sidecar-прокси более существенно влияет на сервисы при обновлении плоскости данных. Процесс обновления подразумевает повторное развертывание всех Pod'ов сервиса, так как без этого нельзя обновить sidecar компонент.

Node-прокси

Node-прокси— это альтернативная архитектура плоскости данных. Вместо того чтобы внедрять прицепной компонент в каждый сервис, mesh-сеть предоставляет отдельный прокси-сервер для каждого узла, который обрабатывает трафик всех сервисов, размещенных на этом узле (рис. 6.22). Среди mesh-сетей, реализующих эту архитектуру, можно выделить Consul Connect (**https://www.consul.io/docs/connect**) и Maesh (**https://containo.us/maesh**). Этот же подход применялся в первой версии проекта Linkerd, но в версии 2 произошел переход на "прицепную" модель.

Если сравнивать с архитектурой sidecar-прокси, node-прокси подход может серьезно повлиять на производительность сервисов. Все сервисы на узле используют один и тот же прокси-сервер, поэтому они не застрахованы от проблемы "шумного соседа", который занимает все сетевые ресурсы. В такой ситуации прокси-сервер может стать узким местом сети.

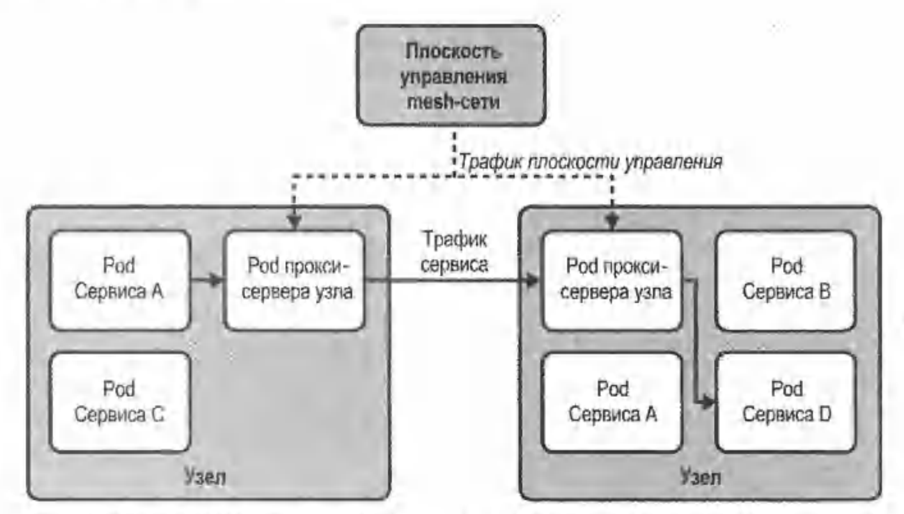


Рис. 6.22. Согласно узловой модели, за обработку трафика всех сервисов на узле отвечает один прокси-сервер mesh-сети

Внедрение mesh-сети

Внедрение mesh-сети может показаться непростой задачей. Стоит ли ее развертывать в уже имеющемся кластере? Как не нарушить выполнение уже запущенных приложений? Как выборочно подключать сервисы для тестирования?

В этом разделе мы исследуем различные факторы, которые следует учитывать при добавлении mesh-сети в свою платформу приложений.

Сосредоточьтесь на одной из характеристик mesh-сети

Один из первых шагов состоит в выборе какой-то определенной характеристики mesh-сети, которой будет отдан приоритет. Это позволит вам сосредоточиться на определенных аспектах реализации и тестирования. В зависимости от ваших требований (с которыми вы уже определились, прежде чем внедрять mesh-сеть, правда?), приоритет, к примеру, может быть отдан TLS. В этом случае основное внимание можно уделить развертыванию инфраструктуры открытых ключей, необходимых для поддержки этой функциональности. Не нужно беспокоиться о настройке средств трассировки или тратить время, выделенное для разработки, на тестирование и администрирование маршрутизации трафика.

Сосредоточившись на какой-то одной фундаментальной характеристике, вы сможете больше узнать о mesh-сети, лучше понять, как она себя ведет на вашей платформе, и получить опыт ее эксплуатации. Почувствовав уверенность, вы сможете при необходимости реализовать другие фундаментальные свойства. Суть в том, что постепенное добавление возможностей дает больше шансов на успех, чем попытка реализовать все сразу.

Где развертывать: в новом или существующем кластере?

В зависимости от жизненного цикла и топологии вашей платформы, у вас может появиться необходимость выбора места развертывания mesh-сети. Вы можете развернуть ее в новом, свежем кластере или добавить ее в уже существующий. По возможности отдавайте предпочтение первому варианту. Это поможет избежать любых потенциальных нарушений в работе приложений, выполняемых в уже имеющемся кластере. Если вы используете временные кластеры, то развертывание mesh-сети в новом окружении должно быть естественным выбором.

Если вам нужно внедрить mesh-сеть в имеющийся кластер, обязательно проведите обширное тестирование в тестовом и отладочном окружениях. Что более важно, обозначьте период, на протяжении которого группы разработчиков смогут поэкспериментировать и проверить работу своих сервисов с mesh-сетью, прежде чем переходить к развертыванию в среде финального тестирования и промышленном окружении. Наконец, предоставьте механизм, который позволяет приложениям присоединяться к mesh-сети по желанию. Зачастую для этого используют аннотацию, которую можно указать в Pod'е. Istio, к примеру, предоставляет аннотацию (`sidecar.istio.io/inject`), которая определяет, должна ли платформа внедрять sidecar-прокси в приложение. Это проиллюстрировано в листинге 6.20.

Листинг 6.20

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
      - name: nginx
        image: nginx
```

Обновление mesh-сети

Предлагая mesh-сеть в рамках своей платформы, вы должны заранее выработать надежную стратегию обновления. Не забывайте, что плоскость данных mesh-сети является одним из важнейших элементов кластера, который соединяет между собой ваши сервисы, в том числе и граничные (независимо от того, используете вы шлюз Ingress mesh-сети или другой контроллер Ingress). Что произойдет, если будет найдена уязвимость, затрагивающая прокси-сервер mesh-сети? Как организовать эффективный процесс обновления? Не начинайте внедрение mesh-сети, пока не разберетесь в этих вопросах и не выработаете хорошо обоснованную стратегию.

Стратегия обновления должна охватывать как плоскость управления, так и плоскость данных. Обновлять плоскость управления менее рискованно, так как плоскость данных mesh-сети должна работать и без нее. Тем не менее, к этому процессу нужно относиться серьезно. Вы должны понимать, совместимы ли версии этих двух плоскостей. По возможности воспользуйтесь моделью канареечных обновлений, которую рекомендует проект Istio (**https://oreil.ly/TZj7F**). Также обязательно анализируйте все изменения, вносимые в пользовательские ресурсы mesh-сети, и пытайтесь понять, имеют ли они какое-либо влияние на ваши сервисы.

Обновление плоскости данных требует больше усилий, учитывая большое число прокси-серверов, развернутых на платформе, и тот факт, что они управляют трафиком сервисов. Когда прокси-сервер работает в sidecar режиме, при его обновлении необходимо заново создать Pod, так как Kubernetes не поддерживает обновление отдельных контейнеров. Причины, стоящие за обновлением плоскости данных, диктуют стратегию ее проведения: сразу или постепенно. С одной стороны, если вы обновляете плоскость данных, чтобы закрыть уязвимость в прокси-сервере, то вам нужно создать заново каждый Pod, входящий в mesh-сеть. Как можно себе представить, это может нарушить работу некоторых приложений. С другой стороны, если обновление мотивировано новыми возможностями или исправлениями ошибок, то новую версию прокси-сервера можно выкатывать по мере создания Pod'ов или перемещения их по кластеру. Эта медленная, менее агрессивная стратегия приводит к тому, что в кластере появляются разные версии прокси-сервера, что может быть допустимо, если это поддерживает mesh-сеть. Независимо от причины обновления, всегда проверяйте новые версии в окружениях для разработки и тестирования.

Также стоит обратить внимание на то, что mesh-сети обычно поддерживают какой-то узкий диапазон версий Kubernetes. Каким образом обновление Kubernetes скажется на вашей mesh-сети? Препятствует ли наличие mesh-сети обновлению Kubernetes сразу после выхода новой версии? Учитывая, что Kubernetes имеет относительно стабильные API-интерфейсы, такой проблемы возникнуть не должно. Но это все же может произойти, и об этом нужно помнить.

Дополнительный расход ресурсов

Один из основных недостатков использования mesh-сети состоит в накладных расходах, которые с этим связаны, особенно в прицепной модели. Как уже упоминалось, mesh-сеть внедряет прокси-сервер в каждый Pod кластера. Для выполнения

своих обязанностей прокси-сервер потребляет ресурсы (процессор и память), которые могли бы быть доступны другим сервисам. Вы должны учитывать этот факт при определении оправданности внедрения mesh-сети. Если ваш кластер размещен в центре обработки данных, дополнительный расход ресурсов, скорее всего, будет приемлемым. Но в граничных развертываниях, где не так много ресурсов, этот фактор может послужить причиной отказа от mesh-сети.

Наверное, еще более важен тот факт, что mesh-сеть увеличивает задержку при взаимодействии сервисов, так как вызовы, которые они делают, проходят через прокси-сервер как на выходе, так и на входе. И хотя в mesh-сетях обычно применяются высокопроизводительные прокси-серверы, они повышают латентность, и вы должны понимать, может ли ваше приложение работать в таких условиях.

В ходе оценивания mesh-сети потратьте время на исследование дополнительных ресурсов, которые она расходует. Или еще лучше, проведите тестирование производительности для своих сервисов, чтобы увидеть, как mesh-сеть ведет себя под нагрузкой.

Центр сертификации для mTLS

Средства идентификации mesh-сети обычно основаны на сертификатах X.509. Ее прокси-серверы используют эти сертификаты для установления между сервисами соединений по mTLS.

Но, прежде чем начать пользоваться этими возможностями mesh-сети, необходимо определиться со стратегией управления сертификатами. Mesh-сеть обычно отвечает за выдачу сертификатов своим сервисам, однако центр сертификации (ЦА) выбираете вы. В большинстве случаев роль ЦС играет самозаверенный сертификат. Но развитые mesh-сети позволяют вам при необходимости задействовать свой собственный ЦС.

Поскольку mesh-сеть управляет взаимодействием между сервисами, использование самозаверенного сертификата видится адекватным решением. ЦС — это фактически аспект реализации, о котором вашим приложениям и их клиентам ничего не известно. Тем не менее, наличие самозаверенного сертификата может быть не одобрено отделом информационной безопасности. Обязательно привлеките этот отдел к процессу внедрения mesh-сети.

Если у вас нет возможности воспользоваться самозаверенным сертификатом для mTLS, вам придется предоставить сертификат и ключ ЦС, с помощью которых mesh-сеть сможет выдавать свои сертификаты. В качестве альтернативы можно интегрировать внешний ЦС, например Vault, если это возможно.

Многокластерная mesh-сеть

Некоторые mesh-сети могут охватывать сразу несколько кластеров Kubernetes. Это нужно для того, чтобы соединить сервисы, размещенные в разных кластерах, с помощью безопасного канала, который с точки зрения приложений выглядит прозрачно. Многокластерные mesh-сети усложняют вашу платформу. Они могут отри-

цательно сказываться как на производительности, так и на отказоустойчивости, и это должны учитывать разработчики. В любом случае, несмотря на внешнюю привлекательность многокластерных mesh-сетей, вам лучше держаться от них подальше, пока вы не научитесь успешно применять mesh-сети в рамках одного кластера.

Резюме

Маршрутизация сервисов — это ключевой аспект создания платформы приложений поверх Kubernetes. Сервисы предоставляют приложениям возможности маршрутизации и балансировки нагрузки на сетевом и транспортном уровнях (L3/L4). С их помощью приложения могут взаимодействовать с другими сервисами в кластере, не беспокоясь о том, что Pod поменяет свой IP-адрес или узел кластера выйдет из строя. Более того, разработчики могут посредством Сервисов NodePort и LoadBalancer предоставлять доступ к приложениям клиентам, находящимся за пределами кластера.

Объект Ingress использует Сервисы для расширения возможностей маршрутизации. С помощью Ingress API разработчики могут маршрутизировать трафик с учетом свойств прикладного уровня, таких как заголовок запроса `Host` или путь, по которому пытается пройти клиент. Ingress API реализуется контроллером Ingress, который необходимо развернуть, прежде чем выделять ресурсы для управления входящим трафиком. После установки контроллер Ingress берет на себя обработку входящих запросов и их маршрутизации в соответствии с конфигурацией Ingress, определенной в API-интерфейсе.

Если у вас есть большое портфолио приложений, основанных на микросервисах, то вашим разработчикам могут быть полезны возможности, предоставляемые mesh-сетью. При наличии mesh-сети сервисы взаимодействуют друг с другом через прокси-серверы, которые дополняют передаваемую ими информацию. Mesh-сети имеют разнообразные возможности, включая управление трафиком, mTLS, управление доступом, автоматический сбор метрик сервисов и др. Как и другие интерфейсы в экосистеме Kubernetes, SMI (Service Mesh Interface — интерфейс mesh-сетей) создан для того, чтобы пользователи платформы могли работать с mesh-сетями, не привязываясь к определенной реализации. Однако, прежде чем внедрять mesh-сеть у себя, убедитесь в том, что у вашей команды достаточно знаний и опыта для эксплуатации еще одной распределенной системы поверх Kubernetes.

ГЛАВА 7

Управление конфиденциальными данными

В любом стеке приложений мы почти наверняка сталкиваемся с конфиденциальными данными, которые требуется хранить в секрете. Обычно мы ассоциируем их с учетной информацией, которая зачастую необходима для доступа к системам внутри или за пределами кластера, таким как БД или очереди сообщений. К конфиденциальным данным также относятся закрытые ключи, которые позволяют нам устанавливать mTLS-соединения с другими приложениями. Такого рода аспекты мы будем рассматривать в *главе 11*. Наличие конфиденциальных данных вынуждает нас принимать во внимание множество эксплуатационных факторов:

- *Политики ротации конфиденциальных данных.* Как долго конфиденциальные данные могут оставаться действительными, прежде чем их нужно будет заменить?
- *Политики ротации ключей (шифрования).* Если предположить, что перед сохранением на диск конфиденциальные данные шифруются на прикладном уровне, как долго ключ шифрования может оставаться действительным, прежде чем его нужно будет заменить новым?
- *Политики хранения конфиденциальных данных.* Какие требования должны быть удовлетворены, чтобы сохранить конфиденциальные данные? Нужно ли их хранить на изолированном оборудовании? Должно ли ваше средство управления конфиденциальными данными интегрироваться с аппаратным модулем безопасности (англ. Hardware Security Module или HSM)?
- *План по исправлению проблем.* Что вы планируете делать в случае, если ваши конфиденциальные данные или ключ(-и) шифрования окажутся скомпрометированы? Может ли ваш план или средства автоматизации выполнить свою задачу, не нарушая работу приложений?

Для начала будет полезно определиться с тем, на каком уровне должен находиться механизм управления конфиденциальными данными, который вы будете предлагать своим приложениям. Некоторые организации предпочитают не делать это на уровне платформы и ожидают, что разработчики будут сами динамически внедрять конфиденциальные данные в свои приложения. Например, если организация использует такую систему как Vault, то приложение может обращаться напрямую к ее API-интерфейсу для идентификации и извлечения соответствующих сведений.

Платформы приложений могут даже предлагать библиотеки для непосредственного взаимодействия с этими системами. Например, у Spring есть проект spring-vault для аутентификации с помощью Vault, извлечения конфиденциальных данных и вне-

дрения их значений прямо в классы Java. И хотя это можно делать на прикладном уровне, многие разработчики пытаются предоставить средства работы с конфиденциальными данными уровня предприятия в виде сервисов платформы. Иногда разработчикам приложений даже не нужно задумываться о том, как эти данные туда попадают или какой внешний провайдер (например, Vault) используется внутри.

В этой главе мы подробно обсудим философию, стоящую за работой с конфиденциальными данными в Kubernetes. Начнем с более низких уровней и постепенно будем продвигаться к API-интерфейсам, с помощью которых Kubernetes делает конфиденциальные данные доступными для приложений. Со многими темами в этой книге представлен целый спектр вопросов и рекомендации, которые, с одной стороны, зависят от того, насколько безопасной вы хотите сделать свою систему с учетом имеющихся инженерных ресурсов и готовности идти на риск, а с другой, какого уровня абстракции вы желаете предоставить разработчикам, которые пользуются вашей платформой.

Углубленная защита

Защита конфиденциальных данных по большому счету сводится к ответу на вопрос, "на что мы готовы пойти, чтобы обеспечить их безопасность". Было бы здорово всегда выбирать самый безопасный вариант, но в реальности мы принимаем разумные решения, которые поддерживают "достаточный уровень безопасности", и затем (в идеале) занимаемся их совершенствованием. Это влечет за собой технический долг и риски (тематический долг), которые непосредственно влияют на нашу работу. Однако нельзя отрицать тот факт, что неправильная оценка "достаточного уровня безопасности" может быстро сделать нас знаменитыми, и не в самом хорошем смысле. В следующих разделах мы поговорим об этих уровнях безопасности и выделим некоторые важнейшие аспекты.

Защита может начинаться буквально на физическом уровне. Яркий пример — компания Google. У нее есть несколько публикаций (**https://cloud.google.com/security/overview/whitepaper**) и даже видеопрезентация на YouTube (**https://oreil.ly/dtHUx**) с описанием подобного подхода к безопасности центров обработки данных. Это включает в себя металлоиндикаторы, шлагбаумы, способные остановить небольшие грузовые автомобили, и несколько уровней охраны здания — все это лишь на входе в ЦОД. Такое внимание к деталям распространяется не только на активное оборудование. Когда срок службы дисков подходит к концу, авторизованный персонал Google обнуляет их содержимое и, возможно, уничтожает их физически. Тема физической защиты интересная, но в этой книге мы не станем углубляться в обеспечение безопасности центра обработки данных, хотя облачные провайдеры предпринимают потрясающие меры для защиты своего оборудования на всех уровнях.

Предположим, что кому-то каким-то образом удалось получить доступ к диску, пока он еще не обнулен или не уничтожен. Большинство облачных провайдеров и ЦОД защищают свои диски за счет шифрования информации, которая на них хранится. Для этого провайдеры могут использовать собственные ключи и/или ключи, предоставленные их клиентами, что делает доступ к расшифрованным данным поч-

ти невозможным. Это идеальный пример углубленной защиты. Мы защищены физически в пределах центра обработки данных, и эта защита распространяется на зашифрованную информацию, хранящуюся на самих дисках, что служит дополнительной страховкой на случай, если злоумышленники проникнут внутрь и попытаются сделать что-нибудь с данными пользователей.

Шифрование дисков

Давайте поближе рассмотрим шифрование дисков. Эту процедуру можно выполнять несколькими способами. В Linux для полноценного блочного шифрования часто используют систему LUKS (Linux Unified Key System). Она работает в связке с подсистемой шифрования dm-crypt, доступной в ядре Linux с версии 2.6. Каждая из таких отдельных систем хранения как vSAN, ceph и Gluster поддерживает один или несколько механизмов шифрования содержимого. Облачные провайдеры применяют по умолчанию разные методы шифрования. Если вы хотите включить шифрование для Elastic Block Storage, вам нужно изучить документацию AWS. В AWS шифрование можно включать по умолчанию, что мы и рекомендуем делать. А вот в Google Cloud содержимое шифруется изначально. Как и в AWS, это можно настроить с помощью сервиса управления ключами (Key Management Service или KMS), который позволяет сконфигурировать параметры шифрования, например, предоставить собственные ключи.

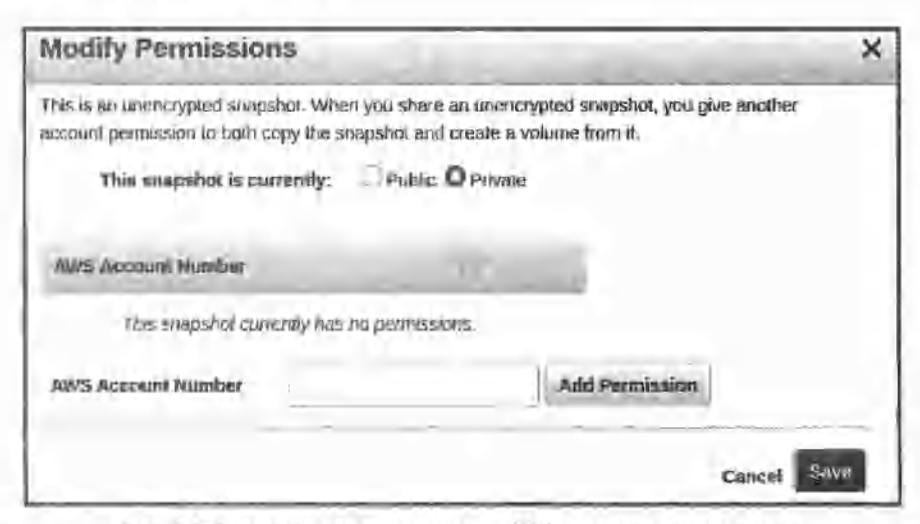


Рис. 7.1. Параметры доступа к снимку AWS позволят другим людям создать том из этого снимка и, возможно, заполучить содержащиеся в нем данные, о чем и говорится в предупреждении

Как бы вы ни доверяли своему облачному провайдеру или работникам ЦОД, мы настоятельно советуем вам шифровать все хранимые данные по умолчанию. Это фактически означает, что данные *хранятся* зашифрованными. В результате не только ограничивается спектр векторов атак, но также обеспечивается определенная страховка на случай ошибок. Например, в мире виртуальных машин создание

снимков систем превратилось в тривиальную задачу. Снимки, как и любые другие файлы можно легко по случайности сделать доступными во внутренней или внешней сети. Следуя духу углубленной защиты, мы должны быть к этому готовы, например, если мы нажмем не ту кнопку в пользовательском интерфейсе или укажем неправильное поле в API, утекшие данные должны быть бесполезными без доступа к закрытому ключу. На рис. 7.1 показано, насколько легко можно переключать эти режимы доступа в пользовательском интерфейсе.

Безопасность во время передачи

Теперь мы лучше понимаем, для чего шифруются хранимые данные. А что можно сказать насчет данных, которые передаются? Пока мы не исследовали архитектуру системы управления объектами Secret в Kubernetes, давайте рассмотрим путь, который проходят данные, которыми обмениваются сервисы. На рис. 7.2 представлены некоторые точки взаимодействия. Стрелки обозначают направление движения конфиденциальных данных (КД) по сети между хостами.

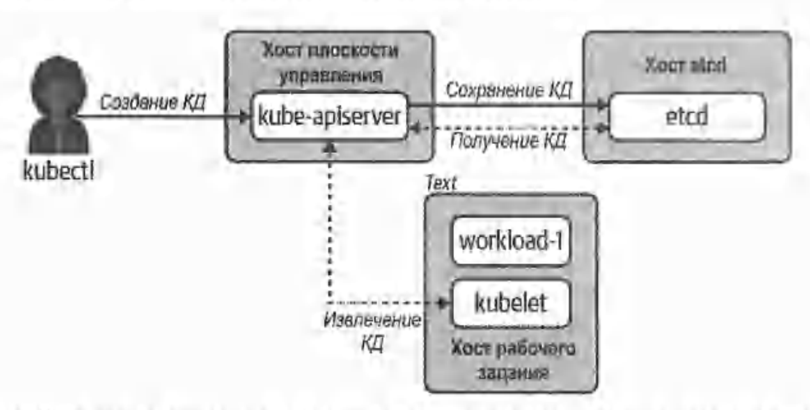


Рис. 7.2. Диаграмма, демонстрирующая этапы, на которых конфиденциальные данные могут быть переданы по сети

На этом рисунке показаны конфиденциальные данные, проходящие по сети к разным хостам. Каким бы надежным ни было шифрование во время хранения, если хотя бы один процесс взаимодействия осуществляется не по TLS, наша информация раскрывается. Как видно из рис. 7.2, это относится к взаимодействию между человеком и системой, между разными системами, при работе с kubectl и при обращении kubectl к API-серверу. Если коротко, то взаимодействие с API-сервером и etcd всегда должно происходить исключительно по TLS. Мы не станем тратить время на обсуждение этого подхода, так как он задан по умолчанию в почти любом режиме установки и начальной конфигурации кластеров Kubernetes. Зачастую вам остается всего лишь указать центр сертификации (ЦС), чтобы генерировать сертификаты. Но помните, эти сертификаты действуют только в рамках системных компонентов Kubernetes. Учитывая это, вам, возможно, и не нужно менять ЦС, который Kubernetes генерирует по умолчанию.

Прикладное шифрование

Прикладное шифрование происходит внутри наших системных компонентов или приложений, выполняемых в Kubernetes. Оно само по себе может иметь несколько уровней. Например, приложение может шифровать данные перед тем, как их сохранять, после чего эти же данные могут шифроваться платформой Kubernetes и сохраняться в etcd, где они будут зашифрованы на уровне файловой системы. Считается, что первые два этапа происходят на "прикладном уровне". Куда ни глянь, везде шифрование!

Данные не всегда шифруются и расшифровываются на таком большом количестве уровней, но случай, когда шифрование на прикладном уровне происходит как минимум один раз, заслуживает дополнительного внимания. Вспомните, что мы обсуждали до сих пор: шифрование по TLS и шифрование хранимых данных. Это уже неплохое начало. Данные шифруются при передаче и сохранении на физический диск. Но что насчет работающей системы? Биты, сохраненные на диск, могут быть зашифрованы, но, если пользователю удастся получить доступ к системе, он, скорее всего, сможет их прочитать! Подумайте о своем настольном компьютере, на котором у вас могут храниться конфиденциальные учетные данные внутри файла с точкой в начале имени (все мы этим грешим). Если я похищу ваш компьютер и попытаюсь извлечь эти данные, подключив диск к своей системе, у меня ничего не получится. Но, если я успешно включу ваш компьютер и *войду в систему от вашего имени*, у меня будет полный доступ к этим данным.

Прикладное шифрование — это процесс шифрования данных в пространстве пользователя с помощью ключа. В упомянутом примере с компьютером я могу воспользоваться GPG-ключом с (надежным) паролем, чтобы зашифровать свой файл. В таком случае им можно будет воспользоваться только после расшифровки. Этот процесс можно автоматизировать с помощью простого скрипта, после чего вам придется иметь дело с *более глубокой* моделью безопасности. Поскольку злоумышленник вошел в систему от вашего имени, его не выручит даже ключ для расшифровки, ведь без пароля это не более чем набор бесполезных битов. То же самое относится и к Kubernetes. В дальнейшем мы будем исходить из того, что в вашем кластере включено шифрование хранимых данных на уровне файловой системы и/или систем хранения, которые использует Kubernetes и для всех компонентов Kubernetes и etcd назначен протокол TLS.

Теперь приступим к исследованию шифрования на прикладном уровне Kubernetes.

Secret API в Kubernetes

Secret API — один из самых востребованных API-интерфейсов в Kubernetes. Наполнять данными объекты Secret можно разными способами, но API-интерфейс дает приложениям возможность взаимодействовать с конфиденциальной информацией согласованным образом. Объекты Secret очень похожи на ConfigMap. У них также есть похожие механизмы, с помощью которых приложения могут их потреб-

лять: посредством переменных окружения или данных о томах. Рассмотрим следующий объект Secret (листинг 7.1).

Листинг 7.1

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  dbuser: aGVwdGlvCg==
  dbkey: YmVhcmNhbm9lCg==
```

Поля dbuser и dbkey в разделе data закодированы в формате base64. То же самое происходит со всеми конфиденциальными данными в Kubernetes. Если вы хотите передать API-серверу незакодированные строки, можете воспользоваться полем stringData, как показано в листинге 7.2.

Листинг 7.2

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  dbuser: heptio
  dbkey: bearcanoe
```

После применения этого ресурса поле stringData будет кодироваться на API-сервере и передаваться агенту etcd. Существует распространенное заблуждение о том, что Kubernetes кодирует эти данные в качестве одной из мер безопасности. Это не так. Конфиденциальные данные могут быть двоичными или содержать всевозможные символы. Чтобы они хранились корректно, их кодируют в base64. По умолчанию ключевым механизмом предотвращения раскрытия объектов Secret является RBAC. Чтобы не создавать новые векторы атаки, необходимо понимать, как работают команды RBAC, относящиеся к объектам Secret:

- get — извлечение данных известного объекта Secret по его имени.
- list — получение списка всех объектов Secret и/или *конфиденциальных данных*.
- watch — отслеживание любых изменений, вносимых в объект Secret и/или *конфиденциальные данные*.

Как можно себе представить, даже такие небольшие ошибки конфигурации RBAC, как выдача пользователю доступа к команде list, могут скомпрометировать все

объекты `Secret` в пространстве имен или, что еще хуже, во всем кластере, если случайно был использован объект `ClusterRoleBinding`. Дело в том, что пользователям зачастую не нужны все эти права, так как RBAC пользователя не определяет, к каким конфиденциальным данным имеет доступ приложение. В целом, за предоставление доступа к конфиденциальным данным для контейнера(-ов) в Pod'е отвечает kubelet. Если коротко, то kubelet делает объект `Secret`, на который ссылается Pod, доступным, используя указанные вами средства (при условии, что этот объект действителен). Открыть приложениям доступ к конфиденциальным данным можно несколькими способами, о которых мы поговорим далее.

Область действия объектов *Secret*

То, что извлечением и внедрением конфиденциальных данных занимается kubelet, крайне удобно. Но в связи с этим напрашивается вопрос: откуда kubelet знает, позволено ли приложению обращаться к этим данным? Kubernetes предлагает приложениям очень простую модель доступа к объектам `Secret`, к лучшему это или к худшему. Объекты `Secret` действуют в пределах пространства имен, т. е. Pod может ссылаться только на те объекты, которые находятся в его пространстве (если только они не реплицируются в рамках всего кластера). Это также означает, что Pod может обращаться к *любым* конфиденциальным данным, доступным в их пространстве имен. Вот одна из причин, почему так важно тщательно продумывать модель разделения пространства имен между приложениями. Если эта модель неприемлема, то можно предусмотреть дополнительные проверки на уровне системы управления допуском, о чем речь пойдет в следующей главе.

Модели потребления объектов *Secret*

У приложения, которое хочет воспользоваться объектом `Secret`, есть несколько вариантов. То, какому из них следует отдать предпочтение, зависит от приложения. Однако у выбранного вами подхода есть свои плюсы и минусы. В следующих разделах мы рассмотрим три механизма потребления конфиденциальных данных в приложениях.

Переменные окружения

Конфиденциальные данные можно внедрять в переменные окружения. В YAML-файле приложения можно указать произвольный ключ со ссылкой на объект `Secret`. Это может оказаться удобной возможностью для приложений, которые переходят на Kubernetes и уже рассчитаны на работу с переменными окружения, так как сокращается объем кода, требующий изменений. Возьмем в качестве примера Pod, приведенный в листинге 7.3.

Листинг 7.3

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    env:
      - name: USER ❶
        valueFrom:
          secretKeyRef:
            name: mysecret ❷
            key: dbuser ❸
      - name: PASS
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: dbkey
```

❶ Ключ переменной окружения, который будет доступен в приложении.

❷ Имя объекта `Secret` в Kubernetes.

❸ Ключ в объекте `Secret`, который должен быть внедрен в переменную `USER`.

Недостаток открытия доступа к конфиденциальным данным в виде переменных окружения — невозможность их динамической перезагрузки. Изменение, внесенное в объект `Secret`, вступит в силу только после повторного создания Pod'а. Это может происходить в результате ручного вмешательства или системных событий, таких, как необходимость перепланирования. Также стоит сказать о том, что некоторые люди считают предоставление конфиденциальных данных в переменных окружения менее безопасным, чем их чтение из подключенных томов. Это спорное мнение, но будет справедливо назвать некоторые распространенные риски утечки информации. В частности, во время анализа процесса или контейнера иногда предоставляется возможность просмотра переменных окружения в виде обычного текста. К тому же, некоторые платформы, библиотеки и языки программирования могут поддерживать отладочные или аварийные режимы, в которых они сбрасывают переменные окружения в журнал. Прежде чем использовать описанный подход, необходимо взвесить потенциальные риски.

Тома

В качестве альтернативы объекты `Secret` можно внедрять с помощью томов. В YAML-файле приложения настраивается том, в котором есть ссылка на конфиденциальные данные. Контейнер, в который эти данные должны внедряться, ссылается на этот том в поле `volumeMount` (листинг 7.4).

Листинг 7.4

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: creds ❷
      readOnly: true
      mountPath: "/etc/credentials" ❸
  volumes: ❶
  - name: creds
    secret:
      secretName: mysecret
```

❶ Тома уровня Pod'а, доступные для подключения. Точка подключения должна ссылаться на указанное имя.

❷ Объект тома, который подключается к файловой системе контейнера.

❸ Указание, в каком месте файловой системы контейнера должна быть доступна точка подключения.

В этом манифесте Pod'а конфиденциальные данные доступны в директории /etc/credentials, а каждая пара "ключ – значение" в объекте `Secret` представлена отдельным файлом:

```
root@nginx:/# cat /etc/credentials/db
dbkey   dbuser
```

Самое важное преимущество рассмотренного подхода состоит в том, что конфиденциальные данные можно обновлять динамически, без перезапуска Pod'а. Когда агент kubelet обнаруживает изменения в объекте `Secret`, он перезагружает этот объект, и тот обновляется в файловой системе контейнера. Необходимо отметить, что в Linux kubelet использует tmpfs, чтобы гарантировать, что конфиденциальные данные хранятся исключительно в оперативной памяти. В этом можно убедиться, заглянув в файл с точками подключения на хосте с Linux:

```
# grep 'secret/creds' secret/creds
tmpfs
/var/lib/kubelet/pods/
e98df9fe-a970-416b-9ddf-bcaff15dff87/volumes/
kubernetes.io~secret/creds tmpfs rw,relatime 0 0
```

Если убрать Pod `nginx` с этого хоста, точка подключения будет удалена. При использовании этой модели особенно важно помнить о том, что конфиденциальные данные *не должны* иметь большой размер. В идеале модель подходит для хранения учетной информации или ключей, и ее нельзя превращать в импровизированную базу данных.

С точки зрения приложения, для обработки изменений, вносимых в объект `Secret`, достаточно просто следить за директорией или файлом и внедрять соответствую-

щие значения. Вам не нужно взаимодействовать с API-сервером Kubernetes и понимать, как он работает. Это идеальный подход, который мы успешно применяем для множества приложений.

Обращение клиентов к API-интерфейсу

Последняя модель потребления конфиденциальных данных не входит в число основных возможностей Kubernetes. Согласно ней, ответственность за взаимодействие с kube-apiserver для извлечения и внедрения объектов `Secret` ложится на само приложение. Существует несколько платформ и библиотек, которые делают взаимодействие вашего приложения с Kubernetes тривиальным. В мире Java эти возможности доступны для приложений на основе Spring в виде библиотеки Spring Cloud Kubernetes, которая берет часто используемый в Spring тип PropertySource и загружает в него объекты `Secret` и/или ConfigMap, подключаясь к Kubernetes.

> **Используйте этот подход с осторожностью**
>
> Объекты `Secret` можно потреблять непосредственно из клиентского приложения, но в целом мы это не приветствуем. Ранее мы уже упоминали о склонности Spring к загрузке конфиденциальных данных, когда речь шла о взаимодействии с API-сервером для получения ConfigMap. Но даже в случае с ConfigMap требовать от каждого приложения обращаться напрямую к API-серверу — не самый оптимальный подход. Если следовать нашей философии, приложение по возможности не должно знать о том, где оно выполняется. Желательно сделать так, чтобы оно могло работать как внутри ВМ, так и в Kubernetes или каком-то другом контейнерном сервисе. У большинства языков и платформ есть средства для чтения переменных окружения и файлов. Kubelet может позаботиться о том, чтобы наши контейнеры получали конфиденциальные данные из этих носителей информации, так зачем же добавлять в наше приложение логику для работы с определенным провайдером, рассчитанную именно на этот случай? Но, даже если не вдаваться в философию, у нас появляется еще один клиент, которому нужно подключаться к API-серверу и подписываться на события. Это не только создает еще одно лишнее соединение, но и вынуждает нас выделить для приложений отдельную служебную учетную запись с сопутствующими ролями для доступа к объекту(-ам) `Secret`. С другой стороны, если делегировать эту работу агенту kubelet, то служебная учетная запись (`default`) не потребуется. Более того, ее можно и нужно отключить!

Итак, мы обсудили потребление конфиденциальных данных на уровне приложений. Теперь пришло время поговорить о том, как эти данные хранятся.

Конфиденциальные данные в etcd

Большинство объектов Kubernetes хранится в etcd, и `Secret` — не исключение. По умолчанию перед их сохранением в etcd на уровне Kubernetes *не* выполняется никакое шифрование. На рис. 7.3 показан путь, который проходят конфиденциальные данные от манифеста к etcd.

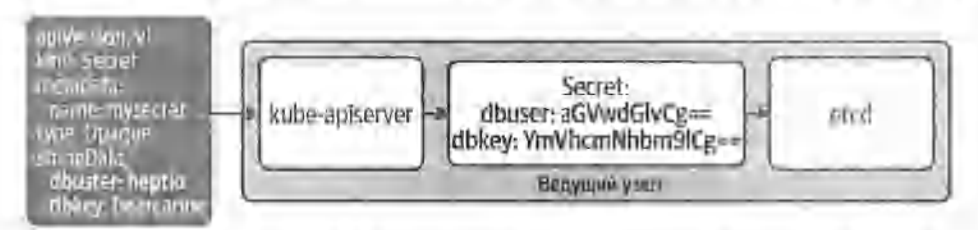


Рис. 7.3 Стандартный маршрут перемещения конфиденциальных данных по Kubernetes (в режиме совместного размещения etcd иногда выполняется на отдельном хосте)

Несмотря на то, что Kubernetes *не* шифрует конфиденциальные данные, это не означает, что их можно прочитать при получении доступа к оборудованию. Как вы помните, хранимые данные могут шифроваться такими методами, как LUKS (Linux Unified Key Setup), в результате чего физический доступ к устройству приводит к получению лишь зашифрованной информации. Для многих облачных провайдеров и корпоративных центров обработки данных это стандартный режим работы. Тем не менее, если получить SSH-доступ к серверу, на котором запущен агент etcd, и к учетной записи, у которой есть привилегии для просмотра файловой системы (или возможность их повышения), это может открыть доступ к конфиденциальным данным.

В некоторых случаях описанная стандартная модель может быть приемлемой. Таким образом, etcd может выполняться за пределами API-сервера Kubernetes и отделяться от него как минимум гипервизором. В результате злоумышленнику пришлось бы получить администраторский доступ к узлу etcd, определить местоположение данных и затем прочитать их из БД etcd. Еще одной точкой входа служит администраторский доступ к API-серверу с последующим нахождением принадлежащих ему и etcd сертификатов, что позволит злоумышленнику выдать себя за API-сервер при взаимодействии с etcd и прочитать конфиденциальные данные. В обоих случаях подразумевается наличие других "дыр" в безопасности. Например, злоумышленнику пришлось бы получить доступ к внутренней сети или подсети, где размещены компоненты плоскости управления. Кроме того, у него должен быть подходящий ключ для входа на узел по SSH. В реальности ошибка в RBAC или взлом приложения с гораздо большей вероятностью могут стать причиной раскрытия конфиденциальных данных.

Чтобы лучше понять природу этой угрозы, рассмотрим пример того, как можно получить доступ к объектам `Secret`. Допустим, злоумышленник зашел по SSH на узел kube-apiserver и получил администраторские привилегии. Он может написать следующий скрипт (листинг 7.5).

Листинг 7.5

```
#!/bin/bash

# Измените значение ниже согласно адресам узлов etcd
ENDPOINTS='192.168.3.43:2379'

ETCDCTL_API=3 etcdctl \
  --endpoints=${ENDPOINTS} \
```

```
  --cacert="/etc/kubernetes/pki/etcd/ca.crt" \
  --cert="/etc/kubernetes/pki/apiserver-etcd-client.crt" \
  --key="/etc/kubernetes/pki/apiserver-etcd-client.key" \
  ${@}
```

Места хранения сертификата и ключа, представленные в этом листинге, заданы по умолчанию при выполнении начальной конфигурации Kubernetes с помощью kubeadm, чем пользуются многие другие инструменты, такие как cluster-api. Конфиденциальные данные etcd хранит в директории конфиденциальные данные в директории /registry/secrets/${NAMESPACE}${SECRET_NAME}. Получение с помощью этого скрипта объекта `Secret` с именем `login1` иллюстрирует листинг 7.6.

Листинг 7.6

```
# ./etcctl-script get /registry/secrets/default/login1

/registry/secrets/default/login1
k8s

v1Secret

login1default"*$6c991b48-036c-48f8-8be3-58175913915c2bB
0kubectl.kubernetes.io/last-applied-configuration{"apiVersion":"v1","data":
{"dbkey":"YmVhcmNhbm9lCg==","dbuser":"aGVwdGlvCg=="},"kind":"Secret",
"metadata":{"annotations":{},"name":"login1","namespace":"default"},
"type":"Opaque"}
z
dbkey
bearcanoe

dbuserheptio
Opaque"
```

В результате мы успешно скомпрометировали объект `login1`.

Хранение незашифрованных объектов `Secret` может быть приемлемым, но многие администраторы платформ на этом не останавливаются. Kubernetes поддерживает несколько моделей шифрования содержимого etcd, что усиливает защиту конфиденциальных данных. В число этих моделей входит шифрование информации до ее попадания в etcd (на уровне Kubernetes) с помощью статического ключа и методом конвертов.

Шифрование с использованием статического ключа

API-сервер Kubernetes поддерживает хранение конфиденциальных данных в зашифрованном виде. Для этого ему нужно предоставить ключ, с помощью которого он будет шифровать все объекты `Secret` перед тем, как записывать их в etcd. На

рис. 7.4 показан путь, который проходит объект `Secret`, при включенном шифровании на основе статического ключа.

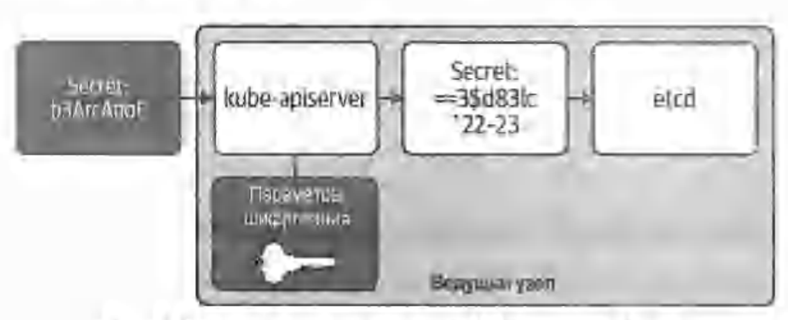


Рис. 7.4 Ключ, с помощью которого API-сервер шифрует конфиденциальные данные перед их сохранением в etcd

Ключ, хранящийся в файле EncryptionConfiguration, служит для шифрования и расшифровки объектов Secret по мере их прохождения через API-сервер. Если злоумышленник получит доступ к etcd, он увидит внутри зашифрованные данные, это означает, что они не скомпрометированы. Ключи можно создавать с использованием различных провайдеров, включая secretbox, aescbc и aesgcm.

У каждого провайдера есть свои плюсы и минусы, и мы советуем вам проконсультироваться со своим отделом безопасности для выбора подходящего варианта. Интересный материал о различных аспектах, касающихся этих провайдеров, можно найти в отчете о проблемах № 81127. Если вашей организации нужно соблюдать такие стандарты, как FIPS (Federal Information Processing Standards — федеральные стандарты обработки информации), ваш выбор должен быть тщательно обоснован. В нашем примере мы будем использовать провайдер шифрования secretbox, который имеет довольно хорошую производительность и степень защиты.

Чтобы настроить шифрование со статическим ключом, мы должны сгенерировать 32-разрядный ключ. Мы используем симметричную модель шифрования, поэтому тот же ключ подходит и для расшифровки. В разных организациях ключи генерируются по-разному. На компьютере с Linux можно легко воспользоваться устройством /dev/urandom, если нас удовлетворяет уровень его энтропии:

```
head -c 32 /dev/urandom | base64
```

На все узлы, на которых выполняется kube-apiserver, нужно добавить объект `EncryptionConfiguration` с содержимым этого ключа. Данный статический файл должен быть добавлен с помощью диспетчера конфигурации такого, как ansible или `KubeadmConfigSpec`, если используется Cluster API. Таким образом ключи можно добавлять, удалять и заменять. В следующем примере (листинг 7.7) подразумевается, что конфигурация хранится в /etc/kubernetes/pki/secrets/encryption-config.yaml.

Листинг 7.7

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
```

```
resources:
  - resources:
    - secrets
    providers:
    - secretbox:
        keys:
        - name: secret-key-1
          secret: u7mcOcHKbFh9eVluB18hbFIsVfwpvgbXv650QacDYXA==
    # identity — это обязательный провайдер (используется по умолчанию)
    - identity: {}
```

Порядок, в котором перечисляются провайдеры, имеет значение: шифрование всегда выполняется с использованием первого ключа, а попытки расшифровки предпринимаются в порядке от первого ключа к последнему. Identity — это провайдер по умолчанию, который оставляет данные в открытом виде. Он должен находиться в самом конце. Если указать его первым, конфиденциальные данные не будут шифроваться.

Чтобы представленная конфигурация вступила в силу, нужно обновить каждый экземпляр kube-apiserver и загрузить в него локально `EncryptionConfiguration`. В файл /etc/kubernetes/manifests/kube-apiserver.yaml можно добавить следующий аргумент:

```
--encryption-provider-config=/etc/kubernetes/pki/secrets/encryption-config.yaml
```

После перезагрузки API-сервера(-ов) это изменение вступит в силу, и конфиденциальные данные перед отправкой в etcd начнут шифроваться. Перезагрузка может происходить автоматически в ответ на изменение файла манифеста, например, когда для выполнения API-сервера используются статические Pod'ы. Но, когда вы закончите экспериментировать, вам рекомендуется этот файл размещать на хостах заранее и следить за тем, чтобы провайдер `encryption-provider` был включен по умолчанию. Файл `EncryptionConfiguration` можно добавить с помощью средств управления конфигурацией таких, как Ansible, или воспользовавшись cluster-api и указав статический файл в kubeadmConfigSpec. Обратите внимание на то, что во втором случае `EncryptionConfiguration` станет частью пользовательских данных, поэтому позаботьтесь о том, чтобы эти данные шифровались! Указать API-серверу флаг `encryption-provider-config` можно путем добавления аргумента `apiServer` внутри ClusterConfiguration (при условии, что вы используете kubeadm). В противном случае обеспечьте наличие этого флага с помощью тех механизмов, которые применяются для запуска сервера.

Чтобы проверить, работает ли шифрование, в API-сервер можно добавить новый объект `Secret`. Допустим, он называется `login2`, тогда для его извлечения можно воспользоваться скриптом из предыдущего раздела (листинг 7.8).

Листинг 7.8

```
# ./etcctl-script get /registry/secrets/default/login2

/registry/secrets/default/login2
k8s:enc:secretbox:v1:secret-key-1:^DH
                                    HN,lU/:L kdR<_h (fO$V
```

```
y.
  r/m
M.jVAGP<%B0kZHY)->q|&c?a\i#xoZsVXd+8_rC ygcj[Mv<X5N):MQ'7t
'pLBxq)b_/+r49 f
                    6(iciQ# f$'.ejbprλ=Cp+R-D%q!r/pbv1_.izyPlQ)!!7@X\0
                                                                   EiLr(dwlS
```


Здесь можно видеть, что данные в etcd полностью зашифрованы. Также обратите внимание на метаданные, в которых указаны провайдер (`secretbox`) и ключ (`secret-key-1`), использованный для шифрования. Для платформы Kubernetes это важно, так как она может работать сразу со многими провайдерами и ключами. Давайте посмотрим, зашифрованы ли объекты, созданные до того, как был указан ключ шифрования; предположим, мы можем запросить объект `login1` и прочитать его содержимое:

```
# ./etcctl-script get /registry/secrets/default/login1

/registry/secrets/default/login1
k8s
```

Здесь мы иллюстрируем два важных факта. Во-первых, объект `login1` *не* зашифрован. Несмотря на наличие ключа шифрования, только новые и измененные объекты `Secret` шифруются с помощью этого ключа. Во-вторых, при повторном прохождении через kube-apiserver связь между провайдером и ключом отсутствует, и попытка шифрования не предпринимается. Второй факт важен, поскольку ключи шифрования настоятельно рекомендуется регулярно менять. Допустим, вы меняете их раз в три месяца. По истечении этого периода мы можем обновить файл EncryptionConfiguation следующим образом:

```
- secretbox:
        keys:
        - name: secret-key-2
          secret: xgI5XTIRQHN/C6mlS43MuAWTSzuwkGSvIDmEcw6DDl8=
        - name: secret-key-1
          secret: u7mcOcHKbFh9eVluB18hbFIsVfwpvgbXv650QacDYXA=
```

Ключ `secret-key-1` *обязательно* нужно оставить. Он больше не будет использоваться для шифрования, но с его помощью будут расшифровываться имеющиеся объекты, которые он зашифровал ранее! Если его удалить, API-сервер не сможет возвращать клиентам такие объекты `Secret` как `login2`. Ключ `secret-key-2` идет первым в списке, поэтому с его помощью шифруются все новые объекты. Когда объект `Secret` обновляется, он заново шифруется с помощью этого нового ключа. Но, пока этого не произошло, исходный ключ может оставаться в списке в качестве резервного варианта для расшифровки. Если вы его удалите, ваши клиенты начнут возвращать следующие ответы:

```
Error from server (InternalError): Internal error occurred: unable to transform
key "/registry/secrets/default/login1": no matching key was found for the
provided Secretbox transformer
```

Совет от авторов: проанализируйте новые векторы атаки

С каждым шагом, направленным на углубление вашей защиты, необходимо понимать, как сместились векторы атаки. Модель шифрования на основе статического ключа, несомненно, безопаснее, чем отсутствие какого-либо шифрования. Но следует учитывать, что ключ шифрования находится на том же хосте, что и API-сервер. Во многих развертываниях Kubernetes kube-apiserver и etcd размещены на одном и том же узле, значит, при наличии администраторского доступа злоумышленник может расшифровать нужные ему данные, хранящиеся в etcd. В идеале не следует полагаться лишь на шифрование хранимых данных с использованием статического ключа. Если такой подход недопустим, вам, возможно, стоит задуматься о применении внешнего хранилища конфиденциальной информации или воспользоваться дополнением KMS. Оба альтернативных подхода рассматриваются в следующих разделах.

Шифрование методом конвертов

Начиная с версии 1.10, Kubernetes поддерживает интеграцию с KMS для шифрования методом конвертов. Этот вид шифрования подразумевает наличие двух ключей: один для шифрования ключа (англ. Key Encryption Key или KEK), а другой для шифрования самих данных (англ. Data Encryption Key или DEK). Ключи KEK хранятся за пределами кластера в KMS и не подвержены риску (если только не скомпрометирован провайдер KMS). С их помощью шифруются ключи DEK, предназначенные для шифрования объектов `Secret`. Каждый объект получает собственный уникальный DEK для шифрования и расшифровки данных. Поскольку ключи DEK шифруются с помощью KEK, они могут храниться вместе с самими данными, благодаря чему API-серверу не нужно знать о большом количестве ключей. С архитектурной точки зрения, процесс шифрования методом конвертов выглядит так, как показано на рис. 7.5.

Этот процесс может варьироваться в зависимости от провайдера KMS, но на рис. 7.5 показан общий принцип работы шифрования методом конвертов. У данной модели есть сразу несколько преимуществ:

- KMS находится вне Kubernetes, что усиливает безопасность за счет изоляции.
- Централизация KEK позволяет легко менять ключи.
- Разделение DEK и KEK означает, что конфиденциальные данные никогда не передаются сервису KMS, и тот о них ничего не знает.
- KMS отвечает только за расшифровку ключей DEK.
- Поскольку ключи DEK шифруются, их можно легко хранить вместе с соответствующими объектами `Secret`, что упрощает управление ключами и в сочетании с данными, которые они шифруют.

Подключаемые модули провайдера имеют вид привилегированного контейнера, который реализует gRPC-сервер, способный взаимодействовать с удаленным сервисом KMS. Этот контейнер выполняется исключительно на ведущих узлах, где присутствует kube-apiserver. Затем, по аналогии с настройкой шифрования в предыдущем разделе, мы должны добавить на ведущие узлы ресурс `EncryptionConfiguration` с параметрами для взаимодействия с подключаемым модулем KMS (листинг 7.9).

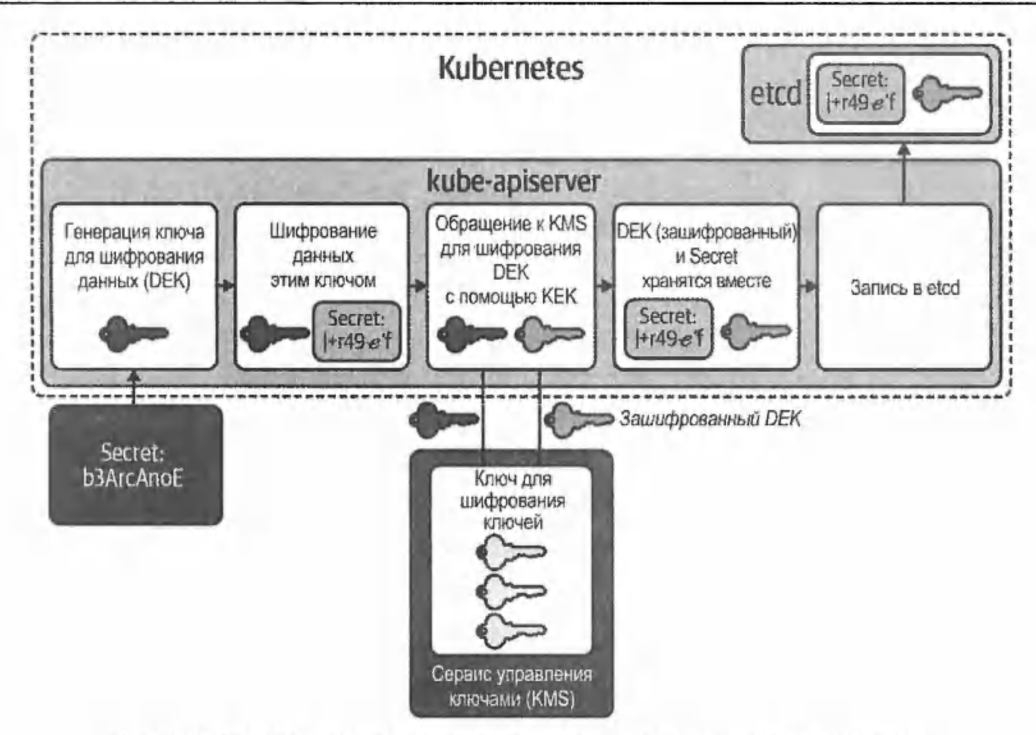


Рис. 7.5. Процесс шифрования конфиденциальных данных методом конвертов. Слой KMS находится за пределами кластера

Листинг 7.9

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
- resources:
- secrets
providers:
- kms:
    name: myKmsPlugin
    endpoint: unix:///tmp/socketfile.sock
    cachesize: 100
    timeout: 3s
# требуется, но не используется при шифровании
- identity: {}
```

Если предположить, что объект `EncryptionConfiguration` хранится на каждом ведущем узле в файле /etc/kubernetes/pki/secrets/encryption-config.yaml, то в число аргументов kube-apiserver нужно добавить следующий:

```
--encryption-provider-config=/etc/kubernetes/pki/secrets/encryption-
config.yaml
```

Изменение данного значения должно привести к перезапуску kube-apiserver. В противном случае это нужно сделать вручную, чтобы изменения вступили в силу.

С точки зрения проектирования, модель жизнеспособна. Однако те немногие реализации подключаемых модулей KMS, которые существуют, недостаточно развиты. На момент написания этих строк наблюдается следующая ситуация. У aws-encryption-provider (AWS) и k8s-cloudkms-plugin (Google) нет помеченных выпусков. Подключаемый модуль kubernetes-kms для Azure имеет существенные ограничения, такие как отсутствие поддержки ротации ключей. Поэтому, если не считать управляемых сервисов вроде GKE, где подключаемый модуль KMS доступен по умолчанию и поддерживается компанией Google, использование описанного механизма может привести к нестабильной работе. Напоследок стоит отметить, что единственным подключаемым модулем KMS, не привязанным к какому-то конкретному облачному провайдеру, был kubernetes-vault-kms-plugin, но этот проект предлагал лишь частичную реализацию, и в итоге его забросили.

Внешние провайдеры

Платформу Kubernetes нельзя назвать хранилищем конфиденциальных данных уровня предприятия. Она предоставляет интерфейс Secret API, который применяется для служебных учетных записей, однако для промышленного использования этого может быть недостаточно. В хранении конфиденциальных данных приложения с помощью такого интерфейса нет ничего принципиально плохого, однако необходимо понимать риски и доступные альтернативы, о чем и шла речь в этой главе до сих пор! Но многие наши клиенты требуют большего, чем то, что может предложить Secret API, особенно те, кто работает в таких сферах, как финансовые услуги. Этим пользователям нужны такие возможности, как интеграция с аппаратным модулем безопасности (англ. Hardware Security Module или HSM) и поддержка нетривиальных стратегий ротации ключей.

Мы обычно советуем начинать с внедрения стандартных механизмов Kubernetes и уже затем думать о том, требуются ли средства углубления безопасности (такие как шифрование). Как уже описывалось в предыдущем разделе, модели шифрования KMS методом конвертов дают довольно хорошую защиту для конфиденциальных данных в etcd. Если нам нужно что-то получше (а зачастую так и есть), мы анализируем существующий инструментарий для управления конфиденциальными данными, с которым знакома команда инженеров. Применение этих инструментов в промышленных условиях может оказаться непростой задачей; то же самое относится к любому сервису, хранящему свое состояние, данные которого должны не только иметь высокую доступность, но и быть защищенными от потенциальных злоумышленников.

Vault

Vault — проект с открытым исходным кодом от HashiCorp. Это, безусловно, самое популярное решение для управления конфиденциальными данными, которое ис-

пользуют наши клиенты. Vault умеет интегрироваться в облачно-ориентированное пространство несколькими путями. Была проделана работа над первоклассной интеграцией в такие платформы, как Spring и собственно Kubernetes. Мы все чаще видим, как Vault размещается внутри Kubernetes и использует TokenReview API для аутентификации запросов к API-серверу. Далее мы рассмотрим два распространенных метода интеграции в Kubernetes: прицепные компоненты в сочетании с внедрением initContainer и более новый подход, интеграцию CSI.

Cyberark

Cyberark — еще одно решение, популярное среди наших клиентов. Эта компания существует довольно давно, и нередко мы видим, что ее продуктами уже пользуются и хотят интегрировать их с Kubernetes. Cyberark предлагает Credential Provider и Dynamic Access Provider (DAP). DAP предоставляет несколько механизмов, с которыми некоторым администраторам Kubernetes имеет смысл интегрироваться. Подобно Vault, DAP умеет взаимодействовать с приложением за счет размещения рядом с ним контейнера initContainer.

Интеграция путем внедрения

Приложения Kubernetes могут извлекать конфиденциальные данные из внешнего хранилища разными способами, которые будут рассмотрены в этом разделе. Мы перечислим их достоинства и недостатки и дадим наши рекомендации. Для каждого из этих методов потребления конфиденциальных данных будет описана соответствующая реализация от Vault.

В этой модели для взаимодействия с внешним хранилищем конфиденциальной информации используется initContainer и/или прицепной контейнер. Обычно объекты Secret внедряются в файловую систему Pod'а, что делает их доступными для всех ее контейнеров. Мы настоятельно рекомендуем применять этот подход, если такое возможно. Его существенное преимущество заключается в том, что хранилище конфиденциальных данных и приложение никак друг с другом не связаны. С другой стороны, это усложняет платформу, так как она теперь должна предоставлять механизм для внедрения объектов `Secret`.

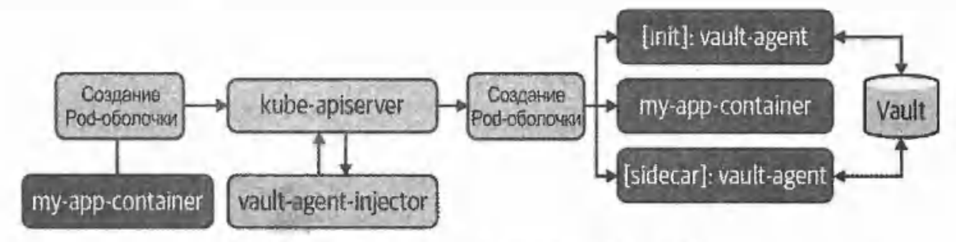


Рис. 7.6. Схема внедрения прицепного контейнера.
Все Pod'ы Vault выполняются в виде sidecar контейнеров вместе с my-app-container

В Vault эта модель реализована с помощью веб-хука MutatingWebhook, направленного к vault-agent-injector. По мере создания Pod'ов vault-agent-injector анализирует аннотации и при необходимости обновляет объекты `Secret`, добавляя initContainer (служит для получения исходных конфиденциальных данных) и прицепной контейнер. Процесс взаимодействия между Pod'ом и Vault продемонстрирован на рис. 7.6.

Конфигурация веб-хука MutatingWebhook, который будет внедрять эти контейнеры, относящиеся к Vault, приведена в листинге 7.10.

Листинг 7.10

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  labels:
    app.kubernetes.io/instance: vault
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/name: vault-agent-injector
  name: vault-agent-injector-cfg
webhooks:
- admissionReviewVersions:
  - v1beta1
  clientConfig:
    caBundle: REDACTED
    service:
    name: vault-agent-injector-svc
    namespace: default
    path: /mutate
    port: 443
failurePolicy: Ignore
matchPolicy: Exact
name: vault.hashicorp.com
namespaceSelector: {}
objectSelector: {}
reinvocationPolicy: Never
rules:
- apiGroups:
  - ""
  apiVersions:
  - v1
  operations:
  - CREATE
  - UPDATE
  resources:
  - pods
  scope: '*'
sideEffects: Unknown
timeoutSeconds: 30
```

MutatingWebhook вызывается в ответ на каждое событие CREATE и UPDATE в контексте Pod'а. Анализу подлежит каждый Pod, но это не означает, что он будет модифицирован, или что в него будет внедрен vault-agent. vault-agent-injector ищет в спецификации каждого Pod'а две аннотации:

- `vault.hashicorp.com/agent-inject: "true"` — заставляет vault-agent-injector добавить контейнер инициализации vault-agent, который извлекает конфиденциальные данные и записывает их в файловую систему Pod'а до запуска других контейнеров.
- `vault.hashicorp.com/agent-inject-status: "update"` — заставляет vault-agent-injector добавить прицепной контейнер vault-agent, который выполняется рядом с приложением. Он обновит объект Secret, если тот изменится в Vault. В этом режиме контейнер инициализации по-прежнему работает. Это необязательный параметр, и без него прицепной контейнер не добавляется.

Когда vault-agent-injector вносит изменения на основе `vault.hashicorp.com/agent-inject: "true"`, добавляется фрагмент кода, приведенный в листинге 7.11.

Листинг 7.11

```
initContainers:
- args:
  - echo ${VAULT_CONFIG?} | base64 -d > /tmp/config.json
  - vault agent -config=/tmp/config.json
  command:
  - /bin/sh
  - -ec
  env:
  - name: VAULT_CONFIG
  value: eyJhd
image: vault:1.3.2
imagePullPolicy: IfNotPresent
name: vault-agent-init
securityContext:
  runAsGroup: 1000
  runAsNonRoot: true
  runAsUser: 100
volumeMounts:
- mountPath: /vault/secrets
  name: vault-secrets
```

Когда vault-agent-injector встречает аннотацию `vault.hashicorp.com/agent-inject-status: "update"`, добавляется следующее:

```
containers:
  #
  # КОНТЕЙНЕР ИСХОДНОГО ПРИЛОЖЕНИЯ УБРАН ДЛЯ КРАТКОСТИ
  #
```

```
- name: vault-agent
  args:
  - echo ${VAULT_CONFIG?} | base64 -d > /tmp/config.json
  - vault agent -config=/tmp/config.json
  command:
  - /bin/sh
  - -ec
  env:
  - name: VAULT_CONFIG
    value: asdfasdfasd
  image: vault:1.3.2
  imagePullPolicy: IfNotPresent
  securityContext:
    runAsGroup: 1000
    runAsNonRoot: true
    runAsUser: 100
  volumeMounts:
  - mountPath: /vault/secrets
    name: vault-secrets
```

Агенты будут извлекать и скачивать конфиденциальные данные в зависимости от аннотаций Pod'а. Например, показанная далее аннотация запрашивает у Vault учетную информацию для доступа к БД:

```
vault.hashicorp.com/agent-inject-secret-db-creds: "serets/db/creds"
```

Конфиденциальное значение сохраняется по умолчанию в том виде, в котором выводятся карты в языке Go. Синтаксически это выглядит следующим образом. Все значения записываются в файл /vault/secrets:

```
key: map[k:v],
key: map[k:v]
```

Чтобы конфиденциальные данные имели оптимальный для потребления формат, Vault поддерживает добавление шаблонов в аннотации Pod. Для этого используется стандартный шаблонизатор языка Go. Например, для создания строки соединения с JDBC к объекту `Secret` под названием `creds` можно применить шаблон, приведенный в листинге 7.12.

Листинг 7.12

```
spec:
  template:
    metadata:
      annotations:
        vault.hashicorp.com/agent-inject: "true"
        vault.hashicorp.com/agent-inject-status: "update"
        vault.hashicorp.com/agent-inject-secret-db-creds: "secrets/db/creds"
        vault.hashicorp.com/agent-inject-template-db-creds: |
```

```
{{- with secret "secrets/db/creds" -}}
jdbc:oracle:thin:{{ .Data.data.username }}/{{ .Data.data.password }}
{{- end }}
```

Основная сложность этой модели состоит в аутентификации и авторизации Pod'а, который выполняет запрос. Vault предоставляет несколько методов аутентификации (**https://www.vaultproject.io/docs/auth**). При работе внутри Kubernetes, особенно если происходит внедрение прицепных контейнеров, аутентификацию имеет смысл сконфигурировать так, чтобы Pod'ы могли предоставлять в качестве идентификаторов токены служебных учетных записей, которые у них уже есть. Настройку такого механизма осуществляют с помощью параметров, приведенных в листинге 7.13.

Листинг 7.13

```
# внутри контейнера Vault

vault write auth/kubernetes/config \
    kubernetes_host="https://$KUBERNETES_PORT_443_TCP_ADDR:443" \ ❶
    kubernetes_ca_cert=@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt \
    token_reviewer_jwt=\
    "$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" ❷
```

❶ Переменная окружения должна присутствовать в Pod'е Vault по умолчанию.

❷ API-сервер использует местоположение токена служебной учетной записи этого Pod'а для аутентификации запросов `TokenReview`.

Когда к Vault поступает запрос на получение объекта `Secret`, проверяется подлинность служебной учетной записи инициатора этого запроса. Для этого Vault обращается к Kubernetes TokenReview API. После выполнения проверки хранилище Vault должно определить, имеет ли эта служебная учетная запись доступ к конфиденциальным данным. Конфигурация и администрирование этих моделей авторизации и их связи со служебными учетными записями должны происходить в рамках Vault. В Vault создается политика следующего вида:

```
# внутри контейнера Vault
vault policy write team-a - <<EOF
path "secret/data/team-a/*" {
  capabilities = ["read"]
}
EOF
```

В результате в Vault будет создана политика под названием `team-a`, которая предоставляет доступ на чтение ко всем конфиденциальным данным внутри secret/data/team-a/:

```
vault policy list
default
team-a
root
```

Осталось только привязать к этой политике служебную учетную запись инициатора запроса, чтобы хранилище Vault могло открыть ему доступ (листинг 7.14).

Листинг 7.14

```
vault write auth/kubernetes/role/database \
    bound_service_account_names=webapp \ ❶
    bound_service_account_namespaces=team-a \ ❷
    policies=team-a \ ❸
    ttl=20m ❹
```

❶ Название служебной учетной записи инициатора запроса.

❷ Пространство имен инициатора запроса.

❸ Привязка этой служебной учетной записи к одной или нескольким политикам.

❹ Период, на протяжении которого должен действовать токен авторизации Vault. По его истечении аутентификация/авторизация выполняются заново.

Этот процесс, рассчитанный на Vault, скорее всего, подходит и для других средств хранения конфиденциальных данных. При работе с системами, не входящими в ядро Kubernetes, вы будете иметь дело с определенными накладными расходами, связанными с интеграцией механизмов идентификации и авторизации для доступа к объектам `Secret`.

Интеграция CSI

Более современный подход к интеграции хранилищ конфиденциальных данных состоит в использовании механизма secrets-store-csi-driver. На момент написания этих строк указанный драйвер является дочерним проектом Kubernetes в рамках kubernetes-sigs. Такой подход позволяет интегрироваться с системами управления конфиденциальной информацией на более низком уровне. В частности, он дает Pod'ам возможность получения доступа к данным за пределами кластера без их внедрения с помощью прицепного контейнера или initContainer. В результате взаимодействие с конфиденциальными данными больше похоже на работу с сервисом платформы, чем на внешнее решение, с которым приложениям нужно интегрироваться. Драйвер secrets-store-csi-driver размещает на каждом хосте Pod с драйвером (в виде DaemonSet), подобно тому, чего можно было бы ожидать от взаимодействия драйвера CSI с провайдером хранилища.

Затем драйвер обращается к провайдеру, ответственному за поиск конфиденциальных данных во внешней системе. В случае с Vault для этого на каждом хосте нужно установить исполняемый файл vault-provider. Он должен находиться в директории, указанной драйвером для точки подключения provider-dir. Этот исполняемый файл может находиться на хосте изначально или устанавливаться с помощью процесса, похожего на DaemonSet (что происходит чаще всего). Общая архитектура будет выглядеть примерно так, как показано на рис. 7.7.

Это довольно новый подход, который выглядит многообещающе, учитывая удобство его использования и возможность инкапсуляции провайдеров хранилищ. Но вместе с тем он создает дополнительные сложности. Например, как происходит

идентификация, когда сам Pod не запрашивает конфиденциальные данные? Это должны определять драйверы и/или провайдеры, так как они выполняют запросы от имени Pod'а. Пока что мы можем рассмотреть основной API-интерфейс, в который входит ресурс `SecretProviderClass`. Чтобы взаимодействовать с внешней системой, такой как Vault, объект `SecretProviderClass` должен выглядеть так, как показано в листинге 7.15.

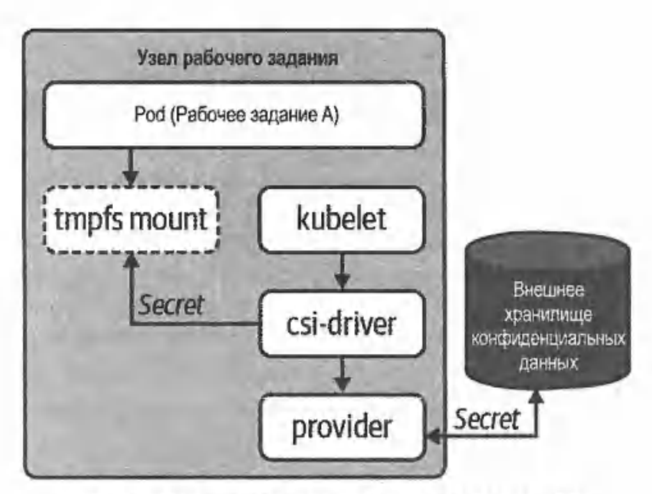


Рис. 7.7. Процесс взаимодействия с драйвером CSI

Листинг 7.15

```
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1
kind: SecretProviderClass
metadata:
  name: apitoken
spec:
  provider: vault
  parameters:
    roleName: "teama"
    vaultAddress: "https://vault.secret-store:8000" ❶
    objects: |
      array:
        - |
          objectPath: "/secret/team-a" ❷
          objectName: "apitoken" ❸
          objectVersion: ""
```

❶ Это местоположение Vault, состоящее из имени Сервиса (`vault`), за которым идет пространство имен `secret-store`.

❷ Это внутренний путь Vault, по которому был записан объект "ключ – значение".

❸ Это сам объект, который нужно искать в `team-a`.

После того как мы создали ресурс `SecretProviderClass`, Pod может его использовать и сослаться на него, как показано в листинге 7.16.

Листинг 7.16

```
kind: Pod
apiVersion: v1
metadata:
  name: busybox
spec:
  containers:
  - image:
    name: busybox
    volumeMounts:
    - name: secrets-api
      mountPath: "/etc/secrets/apitoken"
      readOnly: true
  volumes:
    - name: secrets-api
      csi:
        driver: secrets-store.csi.k8s.com
        readOnly: true
        volumeAttributes:
          secretProviderClass: "apitoken"
```

При запуске Pod'а драйвер и провайдер пытаются получить конфиденциальные данные. Если аутентификация и авторизация на стороне внешнего провайдера пройдут успешно, эти данные появятся в точке подключения тома, как и любой другой объект `Secret` в Kubernetes. Чтобы увидеть, какие команды были посланы провайдеру, можно проанализировать журнальные записи Pod драйвера на узле:

```
level=info msg="provider command invoked: /etc/kubernetes/
secrets-store-csi-providers/vault/provider-vault --attributes [REDACTED]
--secrets [REDACTED] [--targetPath /var/lib/kubelet/pods/
643d7d88-fa58-4f3f-a7eb-341c0adb5a88/volumes/kubernetes.io~csi/
secrets-store-inline/mount --permission 420]"
```

Если подвести итог, то secret-store-csi-drive — это подход, заслуживающий внимания. Со временем, если проект стабилизируется, и провайдеры станут более зрелыми, данная модель может получить популярность среди тех, кто занимается созданием платформ приложений поверх Kubernetes.

Конфиденциальные данные в декларативном мире

В сфере развертывания приложений, непрерывной интеграции и непрерывной доставки наблюдается стремление к переходу на чисто декларативную модель, идентичную той, которая применяется в Kubernetes: вы объявляете желаемое состояние,

и контроллеры постепенно работают над тем, чтобы согласовать его с текущим. Для программистов, занимающихся разработкой и эксплуатацией приложений, это стремление обычно выражается в подходе, известном как GitOps. Главный принцип большинства видов GitOps состоит в использовании одного или нескольких репозиториев git в качестве достоверного источника данных для приложений. Когда изменения фиксируются в какой-либо ветке или теге, они могут быть подхвачены процессами сборки и развертывания, которые зачастую выполняются внутри кластера. Конечной целью этого является получение приложение, способного принимать трафик. Такие модели, как GitOps, подробно рассматриваются в *главе 15*.

Если следовать чисто декларативной модели, конфиденциальные данные создают дополнительные трудности. Конечно, вы можете хранить конфигурацию вместе со своим кодом, но как насчет учетной информации и ключей, которыми пользуется ваше приложение? Что-то нам подсказывает, что запись API-ключа в журнал фиксации изменений может вызвать недовольство у некоторых людей. Но этой проблемы можно избежать несколькими путями. Очевидно, что конфиденциальные данные можно вынести за пределы декларативной модели и затем покаяться за свои грехи перед "богами" GitOps. Еще один способ состоит в "запечатывании" своих конфиденциальных данных так, чтобы доступ к ним не раскрывал ничего важного, и именно это мы исследуем в следующем разделе.

Запечатывание конфиденциальных данных

Как по-настоящему запечатать конфиденциальные данные? В этом нет ничего принципиально нового. Мы можем зашифровать их с помощью ассиметричной криптографии, сохранить их куда-нибудь и не беспокоиться о том, что кто-то их раскроет. Такой подход подразумевает наличие двух ключей: один для шифрования (обычно открытый), а другой для расшифровки (обычно закрытый). Идея состоит в том, что любая информация, созданная с помощью ключа шифрования, не может быть раскрыта, если не скомпрометирован закрытый ключ. Конечно, чтобы сделать эту модель безопасной, нужно позаботиться о многих вещах, таких как выбор шифра, которому можно доверять (гарантия того, что закрытый ключ *всегда* находится в безопасности), и создание политики ротации как ключа шифрования, так и самих конфиденциальных данных. В следующих разделах мы поговорим о том, как это выглядит, когда закрытый ключ генерируется в кластере, и разработчикам выдаются их собственные ключи шифрования, которые они могут применять к своим конфиденциальным данным.

sealed-secret-controller

Bitnami-labs/sealed-secrets — широко распространенный проект с открытым исходным кодом для реализации того, что было описано в предыдущем разделе. Но, даже если вы выберете альтернативный инструментарий или напишете свое решение, ключевые принципы должны остаться примерно такими же.

Основной компонент этого проекта — контроллер sealed-secret-controller, который работает внутри кластера. По умолчанию он генерирует ключи, необходимые для

шифрования и расшифровки. На стороне клиента разработчики используют утилиту командной строки под названием `kubeseal`. Поскольку мы применяем ассиметричное шифрование, утилите `kubeseal` нужно знать только об открытом ключе. Как только разработчики зашифруют с ее помощью свои данные, они больше не смогут расшифровать их напрямую. Для начала развернем этот контроллер в кластере:

```
kubectl apply -f
https://github.com/bitnami-labs/sealed-secrets/releases/\
download/v0.9.8/controller.yaml
```

По умолчанию контроллер создаст для нас ключи шифрования и расшифровки. Но мы можем воспользоваться собственными сертификатами. Открытый сертификат и закрытый ключ хранятся в Kubernetes в объекте `Secret kube-system/sealed-secret-key`. Дальше нужно позволить разработчикам извлечь ключ шифрования, чтобы приступить к работе. Для этого *нельзя* обращаться непосредственно к объекту `Secret`. Контроллер предоставляет конечную точку, с помощью которой можно получить ключ шифрования. Как вы будете к ней обращаться — это уже ваше дело, но у клиентов должна быть возможность взаимодействовать с ней с использованием команды `kubeseal --fetch-cert`, принцип работы которой показан на рис. 7.8.

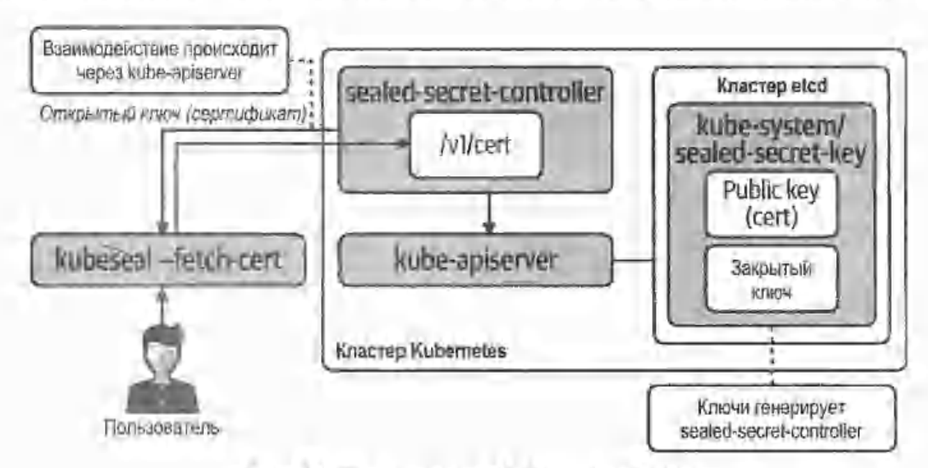


Рис. 7.8. Архитектура sealed-secret-controller

После того, как открытый ключ будет загружен в `kubeseal`, вы сможете генерировать пользовательские ресурсы `SealedSecret`, которые содержат зашифрованные конфиденциальные данные. Эти ресурсы хранятся в etcd. Контроллер sealed-secret-controller делает эти данные доступными в виде стандартных для Kubernetes объектов `Secret`. Чтобы обеспечить корректное преобразование содержимого `SealedSecret` в `Secret`, в объекте `SealedSecret` можно указать шаблон.

Вы можете начать с объекта `Secret`, как обычно (листинг 7.17).

Листинг 7.17

```
apiVersion: v1
kind: Secret
```

```
metadata:
  name: mysecret
type: Opaque
data:
  dbuser: aGVwdGlvCg==
  dbkey: YmVhcmNhbm91Cg==
```

Чтобы его "запечатать", примените к нему команду `kubeseal` и сгенерируйте зашифрованный вывод в формате JSON (листинг 7.18).

Листинг 7.18

```
kubeseal mysecret.yaml

{
  "kind": "SealedSecret",
  "apiVersion": "bitnami.com/v1alpha1",
  "metadata": {
    "name": "mysecret",
    "namespace": "default",
    "creationTimestamp": null
  },
  "spec": {
    "template": {
      "metadata": {
        "name": "mysecret",
        "namespace": "default",
        "creationTimestamp": null
      },
      "type": "Opaque"
    },
    "encryptedData": {
      "dbkey": "gCHJL+3bTRLw6vL4Gf......",
      "dbuser": "AgCHJL+3bT......"
    }
  },
  "status": {

  }
}
```

Объект `SealedSecret` можно разместить где угодно. Данные будут оставаться в безопасности, при условии, что ключ, которым владеет sealed-secret-controller, не скомпрометирован. В этой модели особенно важную роль играет ротация, о чем пойдет речь в следующем разделе.

После применения этого объекта процесс обработки и сохранения данных будет выглядеть так, как показано на рис. 7.9.

Объект `Secret`, созданный контроллером sealed-secret-controller, принадлежит соответствующему пользовательскому ресурсу `SealedSecret`:

```
ownerReferences:
- apiVersion: bitnami.com/v1alpha1
  controller: true
  kind: SealedSecret
  name: mysecret
  uid: 49ce4ab0-3b48-4c8c-8450-d3c90aceb9ee
```

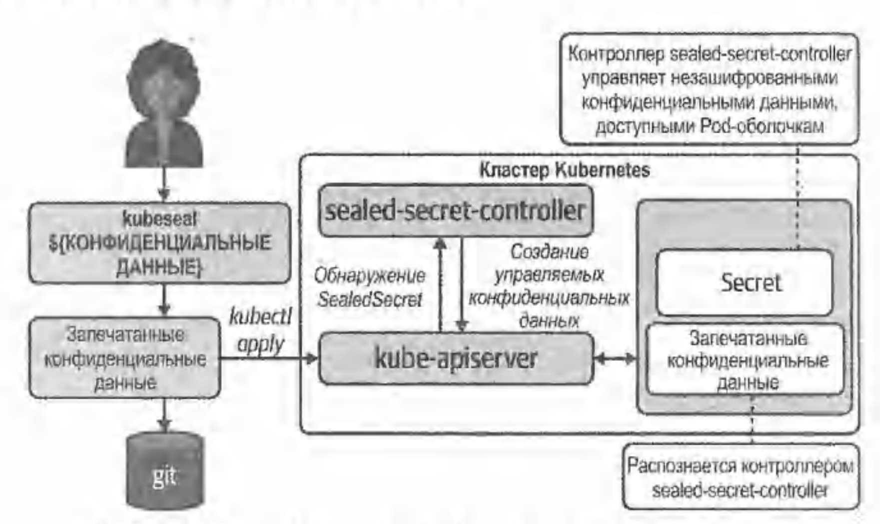


Рис. 7.9. Работа sealed-secret-controller по управлению запечатанными и незапечатанными конфиденциальными данными

Это означает, что при удалении `SealedSecret` сборщик мусора удалит соответствующий объект `Secret`.

Обновление ключей

Если закрытый ключ "утечет" (скажем, из-за некорректной конфигурации RBAC), все конфиденциальные данные, запечатанные с его помощью, следует считать скомпрометированными. Крайне важно, чтобы ключ расшифровки периодически обновлялся, и чтобы вы понимали, что это "обновление" охватывает. По умолчанию этот ключ обновляется раз в 30 дней. Новый ключ не заменяет старый, а добавляется в список имеющихся ключей, способных распечатать данные. Однако новый ключ используется только для шифрования новых данных. Самая важная особенность этого подхода в том, что запечатанные конфиденциальные данные не шифруются заново.

В случае компрометации ключа:

- немедленно выполните ротацию своего ключа шифрования;
- выполните ротацию всех имеющихся конфиденциальных данных;

- помните, что одного лишь повторного шифрования недостаточно. Например, кто-то может без труда заглянуть в историю git, найти там старый зашифрованный ресурс и применить к нему скомпрометированный ключ. В целом для паролей и ключей должны быть предусмотрены стратегии ротации и соответственно обновления.

Один из приемов, который применяется в объекте `SealedSecret`, состоит в использовании пространства имен во время шифрования. Это позволяет добиться изоляции, в рамках которой объект `SealedSecret` принадлежит тому пространству имен, в котором он был создан, и его нельзя просто так перенести в другое пространство. Поведение по умолчанию является наиболее безопасным, и его следует оставить без изменений. Тем не менее, sealed-secret-controller поддерживает настраиваемые политики доступа, о которых можно почитать в документации этого проекта.

Многокластерные модели

Еще один ключевой аспект модели запечатанных конфиденциальных данных связан с методами развертывания в рамках множества кластеров. Во многих из этих методов кластеры считаются временными. В таких случаях использование контроллеров в стиле sealed-secret может быть затруднительным, так как вам необходимо позаботиться о создании уникальных ключей для каждого кластера (если только вы не назначаете им одни и те же закрытые ключи). Кроме того, конечная точка, из которой разработчик должен получить свой ключ (как описывалось в предыдущем разделе), принадлежит уже не одному, а множеству кластеров. Естественно, эта проблема решаема, но над ней стоит подумать.



Работая с клиентами, мы часто сталкиваемся с плохим пониманием того, какие проблемы решает SealedSecret. Бытует мнение, что подход с запечатыванием можно применять в качестве альтернативы шифрованию в etcd или хранилище конфиденциальных данных уровня предприятия, таком как Vault. Но это *не* то, для чего создавался этот подход! Он позволяет нам шифровать и безопасно хранить данные в репозиториях git. Однако закрытый ключ и незашифрованные конфиденциальные данные по-прежнему находятся в Kubernetes. Это означает, что в отсутствие каких-либо шагов, выходящих за рамки стандартного интерфейса Secret API, они будут существовать в незашифрованном виде (на уровне приложений). Нужно четко представлять себе область действия этого и всех других решений, которые мы обсуждали в данной главе!

Рекомендации по работе с конфиденциальными данными

То, как приложение потребляет конфиденциальные данные, во многом зависит от языка программирования и используемых платформ. Несмотря на разнообразие возможных вариантов, существуют общие рекомендации, которые мы бы хотели предложить к рассмотрению разработчикам приложений.

Всегда проводите аудит взаимодействия с конфиденциальными данными

В конфигурации кластера Kubernetes должна быть включена функция аудита. Аудит позволяет определить события, происходящие с заданными ресурсами. Так вы сможете узнать, кто и когда взаимодействовал с тем или иным ресурсом. Если событие связано с изменением данных, аудит поможет определить, что именно изменилось. Аудит событий, относящихся к конфиденциальным данным, играет важную роль при решении проблем с доступом. Подробней об этом можно почитать в разделе документации, посвященном аудиту.

Не раскрывайте конфиденциальные данные

Конфиденциальные данные лучше никогда не раскрывать, но в мультиарендных окружениях Kubernetes необходимо подумать о том, каким образом они могут утечь. Распространенный вид утечек связан со случайной записью в журнал. Например, мы сталкивались с этим несколько раз, наблюдая за тем, как разработчики платформы создают операторы Kubernetes (см. *главу 11*). Эти операторы зачастую работают с конфиденциальной информацией, предназначенной как для внутренних систем, которыми они управляют, так и для внешних, к которым им нужно подключаться. На этапе разработки эту информацию нередко записывают в журнал с целью отладки. Журнальные записи попадают в потоки stdout/stderr, и во многих платформах, основанных на Kubernetes, направляются в систему анализа журналов. Это означает, что они могут проходить через множество окружений и систем в виде обычного текста.

Kubernetes — это в первую очередь декларативная система. Разработчики пишут манифесты, которые вполне могут содержать конфиденциальные данные, особенно в ходе тестирования. Разработчики должны относиться к этому с осторожностью и следить за тем, чтобы тестируемые объекты `Secret` не попали в репозитории с исходным кодом.

Отдавайте предпочтение томам перед переменными окружения

Самые распространенные способы доступа к конфиденциальным данным, предоставляемые платформой Kubernetes, состоят в передаче значений через переменные окружения или тома. В большинстве случаев следует отдавать предпочтение томам. Переменные окружения имеют более высокую вероятность утечки разнообразными путями, например, посредством команды `echo`, выполненной во время тестирования, или из-за того, что платформа автоматически сбрасывает переменные окружения на диск в процессе запуска или сбоя. Но это не означает, что данные проблемы решаются самим фактом использования томов!

Если не брать во внимание безопасность, ключевое преимущество томов с точки зрения разработчиков приложений заключается в том, что они автоматически обновляются. В результате возможна динамическая перезагрузка конфиденциальных

данных, таких как токены. Если же использовать переменные окружения, то изменения вступают в силу только после перезапуска Pod'а.

Делайте так, чтобы ваши приложения не знали о провайдерах хранилищ для конфиденциальных данных

Существует несколько подходов к извлечению и потреблению конфиденциальных данных в приложениях: от обращения к хранилищу в рамках бизнес-логики до предварительной подготовки переменных окружения перед началом работы. Следуя философии разделения ответственности, мы советуем организовать работу с конфиденциальными данными таким образом, чтобы приложению было все равно, кто ими управляет: Kubernetes, Vault или какой-то другой провайдер. Это сделает ваше приложение переносимым, независящим от платформы и упростит взаимодействие с ним. Упрощение происходит благодаря тому, что для извлечения конфиденциальных данных приложение должно уметь взаимодействовать с провайдером и проходить аутентификацию.

Чтобы получить такую реализацию, которой ничего не известно о провайдере, разработчики приложений должны по возможности загружать конфиденциальные данные из переменных окружения или томов. Как уже отмечалось, тома являются наиболее оптимальным вариантом. В этой модели приложение исходит из того, что конфиденциальные данные находятся в одном или множестве томов. Поскольку тома можно обновлять динамически (без перезапуска Pod'а), приложение, которому нужно динамически перезагружать конфиденциальные данные, может следить за файловой системой. Когда потребление происходит в рамках локальной файловой системы контейнера, неважно, кто предоставляет хранилище: Kubernetes или кто-то другой.

Некоторые платформы для разработки приложений, такие как Spring, предоставляют библиотеки для непосредственного взаимодействия с API-сервером и автоматически внедряют конфиденциальные данные и конфигурацию. Какими бы удобными ни были эти механизмы, взвесьте только что рассмотренные факторы и определите, какие подходы принесут вашему приложению больше всего пользы.

Резюме

В этой главе мы исследовали Kubernetes Secret API, методы взаимодействия с конфиденциальными данными, средства их хранения, способы их похищения и некоторые рекомендованные подходы. Обладая этими знаниями, вы должны подумать о том, насколько глубокая защита вас интересует, и затем расставить приоритеты для ее реализации на каждом уровне.

ГЛАВА 8

Управление допуском

В этой книге неоднократно упоминалось о гибкой модульной архитектуре, которая является одной из сильнейших сторон Kubernetes. Параметры по умолчанию можно изменить, дополнить или взять за основу, чтобы предоставить пользователям платформы альтернативные или полнофункциональные возможности. Одна из областей, которая в особенности выигрывает от этой гибкой архитектуры, — управление допуском. Управление допуском заключается в проверке и модификации запросов, направленных к API-серверу Kubernetes *до* их сохранения в etcd. Эта возможность перехватывать объекты с высокой точностью и степенью контроля делает возможным ряд интересных сценариев. Например:

- Запрет на создание новых объектов в пространстве имен, которое в настоящее время удаляется (на этапе уничтожения).
- Запрет на выполнение Pod'ов от имени администратора.
- Гарантия того, что общий объем памяти, занимаемый всеми Pod'ами в пространстве имен, не превысит лимит, заданный пользователем.
- Гарантия того, что правила Ingress не будут случайно переопределены.
- Добавление sidecar контейнера к каждому Pod'у (например, Istio).

Вначале мы дадим общий обзор процесса допуска, через который проходят все запросы к API-серверу. Затем будут рассмотрены контроллеры, входящие в состав Kubernetes. Это встроенные контроллеры допуска, которые делают возможными некоторые из перечисленных сценариев, и которые можно включать и выключать с помощью флагов для API-сервера. В некоторых случаях, когда требуется менее стандартная реализация, можно использовать гибкую модель веб-хуков. Мы посвятим этой модели много времени, так как она предоставляет наиболее мощные и гибкие методы интеграции управления допуском в кластер. В завершение мы поговорим о Gatekeeper — проекте с открытым исходным кодом, обладающем собственной философией. Он реализует модель веб-хуков и предоставляет дополнительные возможности, которыми удобно пользоваться.



Далее в этой главе мы подробно разберем некоторые примеры кода, написанные на языке программирования Go. Kubernetes и многие другие облачно-ориентированные инструменты реализованы на этом языке ввиду его высоких темпов развития, мощных средств конкурентного выполнения и изящного синтаксиса. Для понимания большей части материала, представленного в этой главе, знание Go не требуется (но, если вас интересует инструментарий Kubernetes, мы советуем вам познакомиться с этим языком), и при рассмотрении плюсов и минусов разработки собственных решений (по сравнению с готовыми) будет учитываться необходимость наличия соответствующих навыков.

Цепочка допуска в Kubernetes

Прежде чем приступать к подробному рассмотрению возможностей и принципов работы отдельных контроллеров, давайте сначала исследуем путь, который запросы проходят к API-серверу и обратно (рис. 8.1).

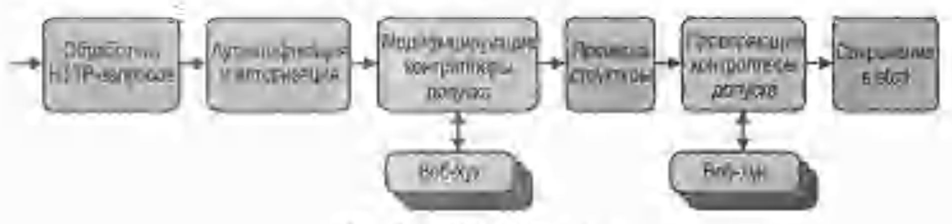


Рис. 8.1. Цепочка допуска

Изначально, когда запросы доходят до API-сервера, они аутентифицируются и авторизуются, чтобы убедиться в том, что клиент ведет себя корректно и способен выполнить запрошенное действие (создание Pod'а в определенном пространстве имен) в соответствии с любыми имеющимися правилами RBAC.

На следующем этапе запросы проходят через контроллеры допуска, которые могут модифицировать запрос. Это могут быть как встроенные контроллеры, так и вызовы внешних (не входящих в Kubernetes) изменяющих веб-хуков (мы рассмотрим их позже в этой главе). Прежде чем передавать ресурс дальше, они могут модифицировать его атрибуты. В качестве примера того, зачем это может понадобиться, рассмотрим контроллер ServiceAccount (встроенный и включенный по умолчанию). Приняв Pod, этот контроллер анализирует его спецификацию, убеждаясь в том, что у него установлено поле `serviceAccount` (SA). Если такого поля нет, он его добавляет и присваивает ему служебную учетную запись, которая используется в текущем пространстве имен по умолчанию (`default`). Он также добавляет объекты `ImagePullSecrets` и `Volume`, чтобы у Pod'а был доступ к своему токену SA (**https://oreil.ly/K6e5E**).

Затем контроллер проверяет, соответствуют ли принятые запросы заранее определенной структуре. Речь идет о наличии обязательных полей. Здесь важен порядок выполнения разных этапов, потому что, прежде чем проверять структуру объекта, модифицирующий контроллер допуска имеет возможность установить нужные поля.

Заключительный этап перед сохранением объекта в etcd состоит в прохождении через проверяющие контроллеры допуска. Это могут быть как встроенные контроллеры, так и внешние (не входящие в состав Kubernetes) проверяющие веб-хуки (мы кратко рассмотрим их позже в этой главе). От модифицирующих контроллеров они отличаются тем, что их возможности ограничиваются лишь приемом или отклонением запроса; они *не* могут его изменить. В отличие от *этапа проверки структуры*, они сверяются не со стандартной спецификацией, а с логикой выполнения.

Примером проверяющего контроллера допуска является NamespaceLifecycle. У него есть несколько обязанностей, связанных с пространствами имен, но нас интересует его способность отклонять запросы на создание новых объектов в пространст-

ве имен, находящемся в процессе удаления. Это поведение продемонстрировано в листинге 8.1.

Листинг 8.1

```
// убеждаемся, что мы не пытаемся создать объекты в удаляемых пространствах имен
if a.GetOperation() == admission.Create {
  if namespace.Status.Phase != v1.NamespaceTerminating {
    return nil ❶
  }

  err := admission.NewForbidden(a, fmt.Errorf("unable to create new content in
  namespace %s because it is being terminated", a.GetNamespace()))
  if apierr, ok := err.(*errors.StatusError); ok {
    apierr.ErrStatus.Details.Causes = append(apierr.ErrStatus.Details.Causes,
    metav1.StatusCause{
      Type:    v1.NamespaceTerminatingCause,
      Message: fmt.Sprintf("namespace %s is being terminated", a.GetNamespace()),
      Field:   "metadata.namespace",
    })
  }
  return err ❷
}
```

❶ Если запрошена операция Create, но пространство имен не удаляется, не возвращаем никаких ошибок. Запрос успешно пройдет через этот контроллер.

❷ В противном случае возвращаем ошибку API-интерфейса, показывающую то, что пространство имен в настоящее время удаляется. Если вернуть ошибку, запрос будет отклонен.



Для успешного прохождения запроса с последующим сохранением объекта в etcd он должен пройти через *все* проверяющие контроллеры допуска. Запрос отклоняется, если его отклонит хотя бы *один* контроллер.

Встроенные контроллеры допуска

Когда платформа Kubernetes только появилась, пользователям было доступно минимальное количество интерфейсов наподобие CNI (Container Network Interface — интерфейс сети контейнеров), которые можно было применять для *подключения модулей* или расширения возможностей. Остальная интеграция с облачными провайдерами и сторонними хранилищами, а также реализация контроллеров допуска были встроены в основную кодовую базу Kubernetes. Со временем разработчики проекта начали стремиться к увеличению числа подключаемых интерфейсов, в результате чего появился интерфейс CSI (Container Storage Interface — интерфейс хранилищ для контейнеров), а сама платформа переориентировалась на внешних облачных провайдеров.

Управление допуском — это одна из тех областей, многие ключевые возможности которых все еще остаются встроенными. В состав Kubernetes входит множество разных контроллеров допуска, которые можно включать и отключать с помощью флагов для API-сервера. Такая модель оказалась проблематичной для тех пользователей, которые применяют Kubernetes в облаке и традиционно не имеют возможности настраивать эти флаги. PodSecurityPolicy (PSP) — пример контроллера, который предоставляет мощные и надежные механизмы безопасности по всему кластеру, но *не* включен по умолчанию, и поэтому пользователям недоступны его возможности.

Однако управление допуском медленно, но верно движется в направлении выноса кода за пределы API-сервера для улучшения расширяемости. Начало этому процессу положило добавление изменяющих и проверяющих веб-хуков. Это два гибких контроллера допуска, которые позволяют указать, что API-сервер должен направлять запросы (соответствующие определенным критериям) и делегировать решения о допуске внешним веб-хукам. Мы подробно обсудим их в следующем разделе.

Очередным шагом на этом пути стал запланированный отказ (**https://github.com/kubernetes/enhancements/issues/5**) от текущего встроенного контроллера PodSecurityPolicy. У него есть несколько потенциальных альтернатив, но, как нам кажется, реализация PSP будет делегирована внешнему контроллеру допуска, так как сообщество продолжает выносить код из ядра проекта. На самом деле мы ожидаем, что та же участь в конечном счете постигнет и другие контроллеры. Им на смену придут либо рекомендуемые сторонние инструменты, либо стандартизированные компоненты, разрабатываемые под эгидой Kubernetes, но за пределами основной кодовой базы, благодаря чему пользователи получат адекватное решение, выбранное по умолчанию, и при необходимости смогут его заменить.



Определенные встроенные контроллеры допуска включены по умолчанию. Это служит *адекватным стандартным выбором*, который должен хорошо работать в большинстве кластеров. Мы не станем воспроизводить подобную конфигурацию, но вы должны позаботиться о том, чтобы нужные вам контроллеры были включены. Также обратите внимание на то, что использование этих возможностей иногда может выглядеть немного запутанным. Для включения дополнительных контроллеров (тех, что не включены по умолчанию), API-серверу необходимо передать флаг `--enable-admission-plugins`, а для *отключения* контроллеров по умолчанию предусмотрен параметр `--disable-admission-plugins`, принимающий список значений.

В официальной документации Kubernetes можно найти много полезных материалов о встроенных контроллерах, поэтому мы не станем углубляться в эту тему. По-настоящему мощными контроллеры допуска делают два особых веб-хука (проверяющий и изменяющий), о которых речь пойдет дальше.

Веб-хуки



Все контроллеры допуска являются частью *важного процесса*, через который проходят запросы, направленные к API-серверу Kubernetes. Их область действия варьируется, поэтому не все запросы могут быть перехвачены, но вы определенно должны это осознавать при включении и/или внедрении этих контроллеров. Это в

особенности касается контроллеров допуска на основе веб-хуков, и для этого есть две причины. Во-первых, они находятся вне ядра Kubernetes, и обращаться к ним нужно по HTTPS, что увеличивает сетевые задержки. Во-вторых, они могут иметь широкий спектр возможностей, в число которых может даже входить обращение к сторонним системам. Вы должны приложить все необходимые усилия для того, чтобы сделать управление допуском как можно эффективней. Чем раньше прерывается процесс обработки запросов, тем лучше.

Веб-хуки — это особый вид контроллеров допуска. Мы можем настроить API-сервер Kubernetes так, чтобы он передавал API-запросы внешнему веб-хуку и принимал от него ответы с решением о том, что с этими запросами делать: пропускать дальше, отклонять или модифицировать. Это чрезвычайно полезная возможность по ряду причин:

- Принимающий веб-сервер может быть написан на любом языке, способном прослушивать HTTPS-запросы. Мы можем использовать веб-платформы, а также знания и опыт, которые могут быть нам доступны, для реализации любой нужной нам логики при принятии решений о допуске.
- Веб-хуки могут работать внутри или за пределами кластера. В первом случае мы можем воспользоваться механизмом обнаружения сервисов и операторами Kubernetes, а во втором у нас, к примеру, есть возможность реализовать нужный нам функционал в виде бессерверной функции, рассчитанной на многократный вызов.
- Для принятия решений о соблюдении политики можно обращаться к системам и хранилищам данных, которые находятся вне Kubernetes. Например, мы могли бы сделать запрос к центральной системе безопасности и проверить, были ли одобрены определенные образы для использования в манифестах Kubernetes.



API-сервер вызывает веб-хуки по TLS, поэтому они должны предоставлять сертификаты, которым доверяет API-интерфейс Kubernetes. Для этого в кластере нередко развертывают Cert Manager и автоматически генерируют сертификаты. Если хуки находятся за пределами кластера, доверенные сертификаты нужно получить либо от публичного корневого ЦС, либо от внутреннего ЦС, о котором знает Kubernetes.

Модель веб-хуков требует определения структуры запросов и ответов, которыми обмениваются API-сервер и сервер веб-хуков. В Kubernetes она описывается в виде объекта `AdmissionReview`, который представляет собой документ JSON со следующей информацией о запросе:

- версия API-интерфейса, группа и тип;
- метаданные, такие как название и пространство имен, а также уникальный идентификатор для сопоставления запроса с ответным решением;
- предпринятая операция (например, Create);
- сведения о пользователе, инициировавшем запрос, включая его принадлежность к той или иной группе;
- является ли этот запрос *пробным* (это важно, в чем вы сможете убедиться позже, когда мы будем обсуждать аспекты проектирования);
- сам ресурс.

Принимающий веб-хук может использовать всю эту информацию для принятия решения о допуске. Как только решение будет принято, сервер должен ответить, послав собственное сообщение `AdmissionReview` (на этот раз с полем `response`). Оно будет содержать:

- уникальный идентификатор из запроса (для сопоставления);
- позволено ли запросу пройти дальше;
- код и сообщение об ошибке, которые можно указать дополнительно.

Проверяющие веб-хуки не могут изменять посланные им запросы, принят или отклонен может быть только исходный объект, что делает их довольно ограниченными. Тем не менее, если вам нужно проверить, соответствует ли объект, примененный к кластеру, стандартам безопасности (наличие определенного идентификатора, отсутствие точек подключения к файловой системе хоста и т. д.), и содержит ли он все необходимые метаданные (внутренние метки группы разработчиков, аннотации и т. д.), они вам пригодятся.

В случае с *изменяющими* веб-хуками структура ответа может включать набор изменений (при необходимости). Это строка, закодированная в base64 и содержащая корректную структуру JSONPatch с изменениями, которые нужно внести в запрос, прежде чем тот будет допущен к API-серверу. Если вас интересует более подробное описание всех полей и структуры объектов `AdmissionReview`, всю необходимую информацию можно найти в официальной документации (**https://oreil.ly/NWagy**).

В качестве примера можно привести простой изменяющий контроллер, который добавляет в объекты Pod или Deployment набор меток с метаданными, относящимися к группе разработчиков или приложению. Более сложный, но в то же время распространенный сценарий, который вам может встретиться, состоит во внедрении прицепного прокси-сервера во многих реализациях mesh-сети. Вот как это работает: mesh-сеть (в данном случае Istio) выполняет изменяющий контроллер допуска, который модифицирует спецификации Pod'ов путем добавления в них прицепного контейнера, участвующего в работе плоскости данных mesh-сети. Это внедрение происходит по умолчанию, но может быть переопределено с помощью аннотаций на уровне пространства имен или Pod'а для дополнительного контроля.

Такая модель дает возможность эффективно расширять возможности объектов Deployment и в то же время скрывать их сложные аспекты, повышая тем самым удобство их использования. Но, как это часто бывает, данное решение имеет и обратную сторону. Недостаток изменяющих контроллеров — их непрозрачность с точки зрения конечного пользователя. К кластеру применяются не те объекты, которые создавались изначально, что может вызвать путаницу, если пользователь не знает о наличии в кластере этих контроллеров.

Настройка контроллеров допуска на основе веб-хуков

Администраторы кластера могут использовать типы `MutatingWebhookConfiguration` и `ValidatingWebhookConfiguration` для задания конфигурации динамических веб-хуков.

В листинге 8.2 приведен пример с кратким описанием интересных нам участков. Некоторые из этих полей будут более подробно рассмотрены в следующем разделе.

Листинг 8.2

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  name: "test-mutating-hook"
webhooks:
- name: "test-mutating-hook"
  rules: ❶
  - apiGroups:   [""]
    apiVersions: ["v1"]
    operations:  ["CREATE"] ❷
    resources:   ["pods"] ❸
    scope:       "Namespaced" ❹
  clientConfig: ❺
    service:
      namespace: test-ns
      name: test-service
      path: /test-path
      port: 8443
    caBundle: "Ci0tLS0tQk...tLS0K" ❻
  admissionReviewVersions: ["v1", "v1beta1"] ❼
  sideEffects: "None" ❽
  timeoutSeconds: "5" ❾
  reinvocationPolicy: "IfNeeded" ❿
  failurePolicy: "Fail" ⓫
```

❶ Правила сопоставления. Какие API/тип/версия/операции нужно отправлять этому веб-хуку.

❷ Операции, которые должны быть инициированы при вызове веб-хука.

❸ На какой тип ресурсов рассчитан этот веб-хук.

❹ Должен ли этот веб-хук должен обрабатывать ресурсы на уровне пространства имен или кластера?

❺ Описывает, как API-сервер должен подключаться к веб-хуку. В данном случае веб-хук находится в кластере и имеет сетевое имя **test-service.test-ns.svc**.

❻ Пакет ЦС, закодированный в формате PEM, с помощью которого будет проверяться подлинность сертификата, принадлежащего серверу веб-хука.

❼ Объявление версий `admissionReviewVersions`, которые поддерживает веб-хук.

❽ Определяет, имеет ли веб-хук внешние побочные эффекты (вызовы к внешним системам или внешние зависимости).

❾ Как долго нужно ждать, прежде чем инициировать `failurePolicy`.

⑩ Может ли этот веб-хук быть вызван повторно (это может произойти после вызова других веб-хуков).

⑪ Каким должен быть сбой веб-хука: *открытым* или *закрытым*. Это важно для безопасности.

Как видно в представленной конфигурации, мы можем крайне точно описать запросы, которые должны перехватывать наши веб-хуки допуска. Например, если нас интересуют только те запросы, которые создают объекты `Secret`, мы можем воспользоваться следующим правилом (цифровые обозначения как в листинге 8.2):

```
# <...опущено...>
rules: ❶
- apiGroups:   [""]
  apiVersions: ["v1"]
  operations:  ["CREATE"] ❷
  resources:   ["secrets"] ❸
  scope:       "Namespaced" ❹
# <...опущено...>
```

Вы также можете сочетать эту конфигурацию с селекторами пространств имен или объектов, что даст вам еще бóльшую гибкость. Таким образом мы сможем указать любое количество нужных нам пространств имен и/или объектов с определенными метками; например, в следующем фрагменте кода мы выбираем только объекты Secret в пространствах имен, у которых есть метка `webhook: enabled`:

```
# <...опущено...>
namespaceSelector:
  matchExpressions:
  - key: webhook
    operator: In
    values: ["enabled"]
# <...опущено...>
```

Аспекты проектирования веб-хуков

При написании и реализации веб-хуков допуска следует учитывать несколько факторов. В следующем разделе мы подробней обсудим их в контексте реальных сценариев использования, но в целом вы должны знать о таких аспектах:

- *Режимы сбоя.* Если веб-хук недоступен или возвращает API-серверу неизвестный ответ, он считается неисправным. В этой ситуации администраторы должны решить, каким должен быть сбой: *открытым* или *закрытым*. Для этого полю `failurePolicy` нужно присвоить либо значение `Ignore` (запрос принимается), либо `Fail` (запрос отклоняется).



Для веб-хуков, имеющих отношение к безопасности, (или важным функциям) лучше всего выбрать значение `Fail`. В остальных случаях может подойти `Ignore` (возможно, в сочетании с контроллером согласования в качестве запасного варианта). При этом следует учитывать рекомендации, перечисленные в пункте "Производительность" этого же списка.

- *Порядок выполнения*. При анализе процесса обработки запросов, направленных к API-серверу, в первую очередь следует обратить внимание на то, что эти веб-хуки будут вызываться (возможно, неоднократно) *перед* проверяющими веб-хуками. Это важно, так как в результате проверяющие веб-хуки (которые могут отклонить запрос в связи с требованиями безопасности) всегда видят *окончательную* версию ресурса перед его применением.

  Изменяющие веб-хуки не всегда вызываются в каком-то определенном порядке и могут выполняться повторно, если их предшественники модифицируют запрос. Такое поведение можно изменить с помощью политики `reinvocationPolicy`, но в идеале веб-хуки следует проектировать так, чтобы порядок вызова не влиял на их работу.

- *Производительность*. Веб-хуки вызываются в рамках важнейшего процесса обработки запросов на пути к API-серверу. Если веб-хук играет важную роль (имеет отношение к безопасности), а сбои сделаны закрытыми (по истечении времени ожидания запрос отклоняется), его нужно проектировать с расчетом на высокую доступность. Как часто отмечает один наш уважаемый коллега (**https://twitter.com/mauilion**), контроллеры допуска могут превратиться в *BaaS* (Bottleneck-as-a-Service — узкое место как услуга[1]), если пользователи будут применять их неосторожно.

  Если веб-хук потребляет много ресурсов и/или имеет внешние зависимости, следует подумать о том, насколько часто он будет вызываться и как его внедрение в процесс обработки запросов повлияет на производительность. В таких ситуациях иногда лучше отдать предпочтение внутрикластерному контроллеру для согласования объектов. При написании конфигураций для веб-хуков пытайтесь как можно сильнее ограничивать их область действия, чтобы они не вызывались без необходимости и не применялись к ресурсам, для которых они не предназначены.

- *Побочные эффекты*. Некоторые веб-хуки могут изменять внешние ресурсы (например, принадлежащие облачному провайдеру) с учетом запросов к API-интерфейсу Kubernetes. Эти веб-хуки должны поддерживать и соблюдать параметр `dryRun`, избегая модификации внешнего состояния, если тот включен. Они обязаны либо объявить о том, что у них нет побочных эффектов, либо учитывать этот параметр за счет использования поля `sideEffects`. Подробней о допустимых значениях этого поля и о том, какое поведение они обусловливают, можно почитать в официальной документации (**https://oreil.ly/8FGic**).

Написание изменяющего веб-хука

В этом разделе мы рассмотрим два подхода к написанию изменяющих веб-хуков допуска. Вначале будет кратко описана реализация на основе обычного HTTPS-обработчика, не относящегося к какому-то конкретному языку программирования.

[1] Шутливый термин. — *Пер*

Затем мы более подробно обсудим реальный пример в контексте проекта controller-runtime, который не входит в основную ветку Kubernetes, но помогает разработчикам создавать компоненты контроллеров.

Оба решения, представленные далее, требуют знания Go (в случае с controller-runtime) или какого-то другого языка программирования. В некоторых случаях это может стать препятствием на пути к созданию контроллеров допуска. Если у ваших разработчиков нет нужных навыков или необходимости в написании собственных веб-хуков, им следует обратиться к заключительным разделам главы, и где предложено решение в виде настраиваемых политик допуска, для создания которых не требуются навыки программирования.

Простой HTTPS-обработчик

Одно из преимуществ управления допуском с помощью веб-хуков заключается в том, что их можно реализовать с нуля на любом языке. Представленные здесь примеры написаны на Go, но для этого подойдет любой язык, способный обрабатывать HTTP-трафик с поддержкой TLS и разбирать документы JSON.

Подобный способ написания веб-хуков наиболее гибкий с точки зрения интеграции с имеющимися технологиями, но при этом требует создания множества высокоуровневых абстракций (хотя в языках с развитыми клиентскими библиотеками для Kubernetes проблема стоит не так остро).

Как уже говорилось во вступительной части раздела, веб-хуки управления допуском принимают и возвращают сообщения, которые получает и отправляет API-сервер. Структура этих сообщений хорошо известна, поэтому мы можем их принимать и модифицировать вручную.

В качестве конкретного примера такого подхода подробно рассмотрим изменяющий веб-хук, назначающий роли IAM для служебных учетных записей в AWS (**https://oreil.ly/rW3ym**), приведенный в листинге 8.3. Он предназначен для внедрения прогнозируемого тома в Pod с токеном служебной учетной записи, который можно использовать для аутентификации в сервисах AWS (в *главе 10* этот сценарий рассматривается более подробно в контексте безопасности).

Листинг 8.3

```
// <...опущено...>
type patchOperation struct { ❶
  Op    string      `json:"op"`
  Path  string      `json:"path"`
  Value interface{} `json:"value,omitempty"`
}

volume := corev1.Volume{ ❷
  Name: m.volName,
  VolumeSource: corev1.VolumeSource{
```

```
    Projected: &corev1.ProjectedVolumeSource{
      Sources: []corev1.VolumeProjection{
        {
          ServiceAccountToken: &corev1.ServiceAccountTokenProjection{
            Audience:          audience,
            ExpirationSeconds: &m.Expiration,
            Path:              m.tokenName,
          },
        },
      },
    },
  },
}

patch := []patchOperation{ ❸
  {
    Op:    "add",
    Path:  "/spec/volumes/0",
    Value: volume,
  },
}

if pod.Spec.Volumes == nil { ❹
  patch = []patchOperation{
    {
      Op:   "add",
      Path: "/spec/volumes",
      Value: []corev1.Volume{
        volume,
      },
    },
  }
}

patchBytes, err := json.Marshal(patch) ❺
// <...опущено...>
```

❶ Определяем структуру `patchOperation`, которая будет преобразована в JSON и отправлена в качестве ответа API-серверу Kubernetes.

❷ Инициализируем структуру `Volume` с использованием подходящих полей для `ServiceAccountToken`.

❸ Создаем экземпляр `patchOperation` с использованием ранее инициализированной структуры `Volume`.

❹ Если в `pod.Spec` нет ключа `Volumes`, создаем его и добавляем ранее инициализированные поля `Volume`.

❺ Создаем объект JSON с содержимым патча.

Обратите внимание, что реализация этого веб-хука допуска содержит некоторые дополнительные возможности, которые тоже расширяют набор изменений (например, за счет добавления переменных окружения), но в данном примере мы их проигнорируем. Применив изменения, мы должны их вернуть внутри объекта `AdmissionResponse` (поле `Patch` в следующем фрагменте кода):

```
return &v1beta1.AdmissionResponse{
  Allowed: true,
  Patch:   patchBytes,
  PatchType: func() *v1beta1.PatchType {
    pt := v1beta1.PatchTypeJSONPatch
    return &pt
  }(),
}
```

Как показывает этот пример, генерация наборов изменений и формирование корректного ответа для API-сервера требует много усилий, даже если использовать библиотеки Kubernetes, доступные для языка Go. При этом мы опустили большой объем вспомогательного кода, необходимого для обработки ошибок, корректного завершения работы, обработки HTTP-заголовков и т. д.

Рассмотренный подход дает максимальную гибкость, но в то же время требует от нас более тесного знакомства с предметной областью и является более сложным в реализации и сопровождении. Большинству читателей уже знакомо это противопоставление гибкости и сложности, поэтому в ходе принятия решения учитывайте свои конкретные рабочие сценарии и то, какими знаниями обладают ваши коллеги.

В следующем разделе мы рассмотрим подход, позволяющий избавиться от большого объема шаблонного и написанного вручную кода за счет использования вспомогательной платформы controller-runtime.

Controller Runtime

В этом разделе мы подробно обсудим проект controller-runtime (**https://github.com/kubernetes-sigs/controller-runtime**), разрабатываемый под эгидой Kubernetes, и посмотрим, какие дополнительные абстракции он предлагает по сравнению со стандартными клиентскими библиотеками для более удобного написания контроллеров допуска. Для наглядности воспользуемся контроллером с открытым исходным кодом, который мы создали для удовлетворения требований сообщества. На его примере будут продемонстрированы некоторые преимущества controller-runtime, а также методики и потенциальные ловушки, которые обсуждались ранее. Несмотря на то, что мы несколько упростили код этого контроллера для краткости, основные принципы остаются в силе.

Проект Kubebuilder

Репозиторий controller-runtime (**https://oreil.ly/sBfth**) содержит примеры, которые могут помочь вам в реализации веб-хуков для встроенных типов (таких как Pod, Deployment и т. д.). Если вы хотите реализовать веб-хуки для пользовательских ресурсов (англ. custom resources или CRD), вам, наверное, больше подойдет проект Kubebuilder

(https://github.com/kubernetes-sigs/kubebuilder), который является более комплексным решением. Kubebuilder использует `controller-runtime` и предоставляет дополнительные средства и вспомогательные компоненты для генерации. Подробней о пользовательских ресурсах и Kubebuilder мы поговорим позже в этой книге.

Тем, кто использует Kubebuilder, этот проект предлагает удобную систему маркеров, которая позволяет генерировать подходящие манифесты для развертывания веб-хуков в кластерах Kubernetes.

Пример:

```
/* +kubebuilder:webhook:path=/infoblox-ipam,mutating=true,failurePolicy=fail,
groups="infrastructure.cluster.x-k8s.io",resources=vspheremachines,
verbs=create,versions=v1alpha3,
name=mutating.infoblox.ipam.vspheremachines.infrastructure.cluster.x-k8s.io */
```

Контроллер, который мы будем рассматривать, представляет собой веб-хук, предназначенный для того чтобы:

1. Следить за объектами Cluster API VSphereMachine.
2. Выделять IP-адрес во внешней системе IPAM (в данном случае Infoblox) с учетом настраиваемого поля.
3. Вставлять выделенный IP-адрес в поле `IPAddrs` в VSphereMachine.
4. Пропускать измененные запросы к API-серверу Kubernetes, чтобы тот выполнял на их основе какие-то действия (посредством контроллера Cluster API) и сохранял их в etcd.

Данный вариант является хорошим кандидатом на создание собственного изменяющего веб-хука (на основе controller-runtime), и на это есть несколько причин:

- *Прежде* чем запрос достигнет API-сервера, мы должны его изменить, добавив в него IP-адрес (иначе произойдет ошибка).
- Мы обращаемся к внешней системе (Infoblox) и, следовательно, можем использовать для взаимодействия ее библиотеку на языке Go.
- Небольшой объем шаблонного кода позволит новым участникам сообщества и/или клиентским разработчикам понять и расширить имеющиеся возможности.



Хотя описанное выходит за рамки *данной* главы, мы снабдили этот веб-хук контроллером, который выполняется внутри кластера. Это важно в случаях, когда ваши веб-хуки взаимодействуют с *внешним* состоянием (в данном примере Infoblox), изменяют его или используют его в качестве зависимости, так как это состояние нужно постоянно согласовывать, а не полагаться на ту его версию, которую вы видели в момент допуска. Данный аспект следует учитывать при создании изменяющих веб-хуков допуска, так как он может усложнить ваши решения, если вам потребуется дополнительный компонент.

Веб-хуки controller-runtime должны реализовывать метод `Handle` со следующей сигнатурой:

```
func (w *Webhook) Handle(
  ctx context.Context,
  req admission.Request) admission.Response
```

Объект `admission.Request`, который принимают веб-хуки, инкапсулирует обычный документ JSON и предоставляет удобный доступ к исходному примененному объекту, выполняемой операции (такой как `CREATE`) и многим другим полезным метаданным:

```
vm := &v1alpha3.VSphereMachine{} ❶
err := w.decoder.DecodeRaw(req.Object, vm) ❷
if err != nil {
  return admission.Errored(http.StatusBadRequest, err) ❸
}
```

❶ Создаем объект `VSphereMachine`.

❷ С помощью встроенного декодера преобразуем исходный объект, пришедший внутри запроса, в объект `VSphereMachine` на языке Go.

❸ Используем вспомогательный метод `Errored`, чтобы сформировать и вернуть ответ с ошибкой, если этап декодирования завершился неудачно.

Перед возвращением ответа объект `vm` из запроса можно модифицировать или проверять любым способом. В следующем примере мы проверяем наличие аннотации `infoblox` (означающей, что наш веб-хук должен выполнить действие) у объекта `VSphereMachine`. Этот шаг важно выполнить в начале веб-хука, так как мы можем предотвратить выполнение любой дальнейшей логики, если нам не нужно выполнять никакого действия. Если аннотации нет, мы используем вспомогательный метод `Allowed`, чтобы как можно быстрее направить API-серверу неизмененный объект. Как уже отмечалось ранее в разделе "Аспекты проектирования веб-хуков" данной главы, веб-хуки являются частью важного процесса обработки API-запросов, и любые выполняемые ими действия должны быть настолько быстрыми, насколько это возможно:

```
if _, ok := vm.Annotations["infoblox"]; !ok {
  return admission.Allowed("Nothing to do")
}
```

Если предположить, что мы *должны* обработать этот запрос, и что приведенная ранее логика не выполняется, мы извлекаем IP-адрес из `Infoblox` (здесь не показано) и записываем его непосредственно в объект `vm`:

```
vm.Spec.VirtualMachineCloneSpec.Network.Devices[0].IPAddrs[0] = ipFromInfoblox ❶
marshaledVM, err := json.Marshal(vm) ❷
if err != nil { ❸
  return admission.Errored(http.StatusInternalServerError, err)
}
return admission.PatchResponseFromRaw(req.Object.Raw, marshaledVM) ❹
```

❶ Присваиваем значение полю `IPAddrs` объекта `vm` и тем самым его *изменяем*.

❷ Преобразуем объект `vm` в документ JSON, готовый к отправке API-серверу.

❸ Если преобразование завершится неудачно, мы воспользуемся вспомогательным методом `Errored`, который уже применялся нами ранее.

❹ Еще один вспомогательный метод, `PatchReponseFromRaw`, возвращает ответ. Позже мы обсудим это более подробно.



Бывают ситуации, когда нам хочется и/или нужно перехватывать запросы `DELETE`, направленные к API-серверу. Это, к примеру, может потребоваться для очистки какого-то внешнего состояния, которое привязано к ресурсам кластера. Это *можно* сделать и в веб-хуке, но здесь следует учитывать, является модель сбоев открытой или закрытой, и какими рисками чревато наличие несогласованного состояния, если модель открытая. В идеале, чтобы гарантировать успешную очистку, логика удаления должна быть реализована с помощью метода завершения (**https://oreil.ly/Y1iGD**) и пользовательского контроллера, который работает в кластере.

В приведенном фрагменте кода мы видим еще один вспомогательный метод из контроллера controller-runtime, `PatchResponseFromRaw`. Он автоматически вычисляет разницу между исходным объектом и тем, который мы модифицируем, и оформляет ее в виде корректно сериализованного ответа JSONPatch, который затем отправляется. Если сравнивать с ручным подходом, рассмотренным в предыдущем разделе, это отличный способ избавиться от некоторого шаблонного кода и сделать наш контроллер более лаконичным.

В случае с простым проверяющим веб-хуком можно также воспользоваться такими вспомогательными методами, как `admission.Allowed()` и `admission.Denied()`, которые можно применять после обработки запроса.



Если мы изменяем внешнее состояние в рамках контроллера допуска, то нам нужно учитывать и проверять параметр `req.DryRun`. Если он установлен, пользователь выполняет лишь пробную операцию, холостой запрос, и в этом случае мы должны позаботиться о том, чтобы наш контроллер *не* модифицировал внешнее состояние.

Controller-runtime служит очень прочным фундаментом для создания контроллеров допуска, позволяя нам сосредоточиться на логике, которую мы хотим реализовать с минимальным объемом шаблонного кода. Тем не менее, это требует навыков программирования, а логика допуска в коде контроллера становится менее ясной, что может привести к результатам, которые для конечного пользователя будут неожиданными и сбивающими с толку.

В следующем разделе мы рассмотрим новую модель, которая централизует логику политики и предлагает стандартный язык описания правил. Инструменты, появляющиеся в этой области, стремятся совместить в себе гибкость пользовательских контроллеров и отличные, удобные в использовании возможности, ориентированные на менее технически подкованных администраторов и/или конечных пользователей.

Системы с централизованными политиками

Мы уже обсудили ряд разнообразных методов реализации и конфигурации контроллеров допуска. У каждого из них есть свои плюсы и минусы, которые нужно учитывать, если вы собираетесь ими пользоваться. В этом заключительном разделе мы поговорим о новой модели, состоящей в централизации логики политик и использовании стандартизированного языка для описания правил допуска/отклонения. У такой модели есть два важных преимущества:

- Для создания контроллеров допуска не требуются навыки программирования, так как правила могут быть выражены на специальном языке описания политик

(а не на языке программирования общего назначения). Это также означает, что для внесения изменений в логику контроллера его не нужно каждый раз пересобирать и заново развертывать.

- Политики и правила хранятся в едином месте (в большинстве случаев в самом кластере), где их можно просматривать, редактировать и анализировать.

Данный подход развивается и реализуется в нескольких инструментах с открытым исходным кодом и обычно состоит из двух компонентов:

- Язык политик/запросов, способный описывать условия, при которых объект должен быть принят или отклонен.
- Контроллер, размещенный в кластере и отвечающий за управление допуском. Его обязанность состоит в том, чтобы проверять объекты, направленные к API-серверу, на соответствие политикам/правилам и принимать решение об их допуске или отклонении.

Далее мы сосредоточимся на самой популярной реализации этой модели централизованных политик, Gatekeeper (**https://github.com/open-policyagent/gatekeeper**), хотя другие инструменты, такие как Kyverno (**https://kyverno.io**), тоже набирают обороты. Проект Gatekeeper основан на более низкоуровневом инструменте под названием Open Policy Agent (OPA). OPA — это подсистема с открытым исходным кодом, которая применяет политики, написанные на языке Rego, к потребленным документам JSON и возвращает результат.



Rego — это декларативный язык запросов, который используется в Open Policy Agent. Он был создан авторами OPA и является языком описания политик в этой подсистеме. Он не задумывался как язык программирования общего назначения, и его основная задача состоит в создании запросов и выполнении операций со структурами данных. Благодаря такому подходу он имеет довольно лаконичный синтаксис, но чтение и написание кода на этом языке может поначалу вызывать сложности. Мы не станем рассматривать синтаксис Rego в этой книге, но те, кто хочет его изучить и проверить свои знания, могут воспользоваться бесплатным порталом онлайн-обучения по адресу **https://acad emy.styra.com/courses/opa-rego**.

Приложение, использующее OPA, может принимать на основе полученных результатов решения о том, как действовать дальше (т. е. решение о выборе политики). Из этой главы нам уже известно о том, что у Kubernetes есть стандартная процедура отправки запросов и получения ответов с решениями о допуске, поэтому на первый взгляд данный подход может показаться многообещающим. Однако сама по себе подсистема OPA не ориентирована ни на какую платформу или контекст, она просто работает с JSON, позволяя описывать политики. Нам нужен контроллер, который будет служить интерфейсом между OPA и Kubernetes. Эту роль может взять на себя инструмент Gatekeeper, который к тому же предоставляет дополнительные возможности по созданию шаблонов и расширению механизмов Kubernetes, помогая администраторам платформы создавать и применять политики. Gatekeeper развертывается в кластере в виде контроллера допуска, давая возможность пользователям писать на Rego правила для принятия решений о допуске ресурсов, которые применяются к кластеру.

Gatekeeper делает возможной модель, в которой администраторы кластера могут создавать и предоставлять готовые шаблоны политик в виде пользовательских ресурсов `ConstraintTemplate`. Эти шаблоны создают новые CRD для определенных ограничений, способные принимать пользовательские параметры (подобно функции). Описанный подход весьма эффективен, так как конечные пользователи могут создавать экземпляры ограничений с собственными значениями, которые затем будут использоваться инструментом Gatekeeper в процессе управления допуском запросов в кластер.



В настоящее время Gatekeeper использует по умолчанию открытую модель сбоев. Вам нужно об этом знать по нескольким причинам, которые будут описаны позже в этом разделе. Это может иметь серьезные последствия для безопасности, и вы должны тщательно взвесить все плюсы и минусы каждого подхода (описанные в этой главе и большой части официальной документации), прежде чем приступать к реализации этих решений в реальных условиях.

В реальных проектах нам часто приходится реализовывать правило, которое не дает разработчикам приложений возможность переопределять существующие ресурсы Ingress. Это требуется в большинстве кластеров Kubernetes, и некоторые контроллеры Ingress (такие, как Contour) сами предоставляют соответствующие механизмы. Но, если ваш инструментарий это не поддерживает, вы можете воспользоваться Gatekeeper, чтобы обеспечить соблюдение данного правила. Это один из нескольких сценариев, которые собраны в библиотеку распространенных политик в официальной документации Gatekeeper (**https://oreil.ly/LINGy**).

В этой ситуации принятие решения о политике должно основываться на данных, *существующих вне объекта*, который применяется к кластеру. Нам нужно обратиться к Kubernetes напрямую, чтобы узнать, какие ресурсы Ingress уже существуют, и чтобы получить возможность проанализировать связанные с ними метаданные и сравнить их с теми объектами, которые применяются.

Давайте рассмотрим еще более сложный пример, основанный на этих идеях, и разберем реализацию каждого ресурса. В данном случае мы аннотируем пространство имен с помощью регулярного выражения и сделаем так, чтобы ему соответствовали любые объекты `Ingress`, применяемые в этом пространстве. Мы уже упоминали о том, что для принятия решений о политике нам нужно сделать информацию о кластере доступной для Gatekeeper. Для этого мы определим конфигурацию, в которой укажем, какие ресурсы Kubernetes должны синхронизироваться с кэшем Gatekeeper, чтобы предоставить источник данных, к которому можно обращаться с запросами (листинг 8.4).

Листинг 8.4

```
apiVersion: config.gatekeeper.sh/v1alpha1
kind: Config
metadata:
  name: config
  namespace: "gatekeeper-system"
```

```
spec:
  sync: ❶
syncOnly:
  - group: "extensions"
    version: "v1beta1"
    kind: "Ingress"
  - group: "networking.k8s.io"
    version: "v1beta1"
    kind: "Ingress"
  - group: ""
    version: "v1"
    kind: "Namespace"
```

❶ В разделе `sync` указаны все ресурсы Kubernetes, которые должен кэшировать Gatekeeper, чтобы помогать нам в принятии решений о политике.



Этот кэш существует для того, чтобы Gatekeeper не нужно было в дальнейшем обращаться к API-серверу Kubernetes за необходимыми ресурсами. С другой стороны, возникает риск, что Gatekeeper начнет принимать решения на основе *неактуальных* данных. Для борьбы с этой проблемой существует процедура аудита, которая периодически применяет политики к существующим ресурсам и записывает нарушения в поле `status` каждого ограничения. Вы должны следить за этими полями, чтобы не оставлять без внимания нарушения, которые могут возникать в том числе из-за чтения устаревшего кэша.

После применения этой конфигурации администратор может создать объект `ConstraintTemplate`. Этот ресурс определяет основное содержимое политики и все входные параметры, которые могут быть предоставлены/переопределены администраторами или другими пользователями кластера (листинг 8.5).

Листинг 8.5

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: limitnamespaceingress
spec:
  crd:
    spec:
      names:
        kind: LimitNamespaceIngress
        listKind: LimitNamespaceIngressList
        plural: limitnamespaceingresss
        singular: limitnamespaceingress
      validation:
        # Поле `parameters` для указания конфигурации
        openAPIV3Schema:
          properties: ❶
            annotation:
              type: string
```

```
  targets: ❷
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package limitnamespaceingress
        violation[{"msg": msg}] {
          cluster := data.inventory.cluster.v1
          namespace := cluster.Namespace[input.review.object.metadata.namespace]
          regex := namespace.metadata.annotations[input.parameters.annotation]
          hosts := input.review.object.spec.rules[_].host
          not re_match(regex, hosts)
          msg := sprintf("Only ingresses matching %v in namespace %v allowed",
            [regex ,input.review.object.metadata.namespace])
        }
```

❶ Раздел `properties` определяет входные параметры, которые будут доступны для внедрения в политику Rego при каждом создании правила.

❷ Раздел `targets` содержит код Rego для правил нашей политики. Мы не станем углубляться в синтаксис Rego, но обратите внимание, что ссылки на входные параметры выглядят как `input.parameters.<название_параметра>` (в данном случае `annotation`).

Входной параметр `annotation` дает возможность пользователю указать имя определенной аннотации, из которой Gatekeeper должен взять регулярное выражение. Rego не распознает нарушение, если какая-либо инструкция вернет `False`. В данном случае мы убеждаемся в том, что хосты *соответствуют* регулярному выражению, поэтому, чтобы не произошло нарушения, нам нужно инвертировать `re_match()` с помощью оператора `not`; таким образом, положительное совпадение не приведет к нарушению, и запрос *сможет* пройти процесс допуска.

В конце мы создаем экземпляр представленной политики и конфигурируем с его помощью Gatekeeper, чтобы тот применял его к определенным ресурсам в процессе управления допуском. Объект `LimitNamespaceIngress` указывает на то, что правило должно применяться для объектов `Ingress` в обеих группах, перечисленных в поле `apiGroups`, и назначает `allowed-ingress-pattern` в качестве аннотации, которая должна анализироваться на предмет наличия регулярного выражения (это был настраиваемый входной параметр):

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: LimitNamespaceIngress
metadata:
  name: limit-namespace-ingress
spec:
  match:
    kinds:
      - apiGroups: ["extensions", "networking.k8s.io"]
        kinds: ["Ingress"]
  parameters:
    annotation: allowed-ingress-pattern
```

Наконец, пользовательская аннотация и регулярное выражение применяются к самому объекту Namespace. Здесь в поле allowed-ingress-pattern указано регулярное выражение \w\.my-namespace\.com:

```
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    # Обратите внимание на экранирование спецсимволов в регулярном выражении
    allowed-ingress-pattern: \w\.my-namespace\.com
  name: ingress-test
```

На этом подготовительный этап завершен. Мы можем приступать к добавлению объектов Ingress, и правила, которые мы сконфигурировали, будут их проверять и либо разрешать их сохранение/создание, либо нет (листинг 8.6).

Листинг 8.6

```
# ОТКЛОНЯЕТСЯ, так как хост не соответствует вышеприведенному регулярному
выражению
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-1
  namespace: ingress-test
spec:
  rules:
  - host: foo.other-namespace.com
    http:
      paths:
      - backend:
          serviceName: service1
          servicePort: 80
---
# ПРИНИМАЕТСЯ ввиду соответствия регулярного выражения
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-2
  namespace: ingress-test
spec:
  rules:
  - host: foo.my-namespace.com
    http:
      paths:
      - backend:
          serviceName: service2
          servicePort: 80
```

Второй объект `Ingress` будет принят, так как `spec.rules.host` соответствует регулярному выражению, указанному в аннотации `allowed-ingress-pattern` для пространства имен `ingress-test`. А вот первый объект не соответствует, что приводит к ошибке:

```
Error from server ([denied by limit-namespace-ingress] Only ingresses with
host matching \w\.my-namespace\.com are allowed in namespace ingress-test):
error when creating "ingress.yaml": admission webhook "validation.gatekeeper.sh"
denied the request: [denied by limit-namespace-ingress] Only ingresses with host
matching \w\.my-namespace\.com are allowed in namespace ingress-test
```

Сильные стороны Gatekeeper:

- Расширяемая модель `ConstraintTemplate` позволяет администраторам определять общие политики и распространять или повторно использовать их в виде библиотек внутри организации.
- Вам не нужно иметь опыта работы ни с какими языками программирования помимо Rego, что снижает порог вхождения для проектирования и создания политик.
- Технология OPA, на которой основан Gatekeeper, довольно зрелая и имеет хорошую поддержку со стороны сообщества. Gatekeeper — относительно новый проект, но его поддержка вышла на высокий уровень уже на начальных этапах развития.
- Объединение всех механизмов, обеспечивающих соблюдение политик, в один контроллер допуска, дает возможность обращаться к единому централизованному журналу аудита, что важно во многих средах, на которые распространяются нормативно-правовые требования.

Главным недостатком Gatekeeper является то, что в настоящее время он не способен *изменять* запросы. И хотя он поддерживает внешние источники данных (посредством разнообразных методов), их реализация может быть громоздкой. В будущем эти проблемы будут обязательно решены, но, если у вас есть жесткие требования в указанных областях, то вам, скорее всего, придется реализовать одно из решений на основе собственного контроллера допуска, как было описано в предыдущих разделах.

Последний фактор, который следует учесть при использовании Gatekeeper (и любого другого контроллера допуска общего назначения) состоит в том, что спектр запросов, перехватываемых этими инструментами, будет, скорее всего, очень широким. Это делает их полезными, так как вы можете писать правила, охватывающие много разных объектов, и сам контроллер должен содержать надмножество прав доступа, чтобы иметь возможность перехватывать эти запросы. Но также могут появиться некоторые неприятные последствия:

- Как уже упоминалось, эти инструменты являются частью важного процесса. Если контроллер и/или ваша конфигурация имеют программную ошибку или какую-то другую проблему, это может привести к широкомасштабному сбою.
- В продолжение предыдущего пункта, ввиду того что контроллеры перехватывают запросы к *плоскости управления*, может случиться так, что вы, как адми-

нистратор, *потеряете возможность* выполнения шагов, направленных на исправление ситуации. Это особенно актуально для ресурсов, которые играют важную роль в *работе* кластера и/или являются его неотъемлемой частью (например, для сетевых ресурсов и т. д.).

- Широкий спектр охвата требует, чтобы за сервером контроллера/политики допуска была закреплена общая политика RBAC. Если в этом ПО есть какая-то уязвимость, возникают широкие возможности для выполнения деструктивных действий.



Не следует настраивать веб-хуки допуска для перехвата ресурсов в пространстве имен `kube-system`. Объекты этого пространства зачастую необходимы для работы кластера, и случайное их изменение или отклонение может привести к серьезным проблемам.

Резюме

В данной главе мы обсудили множество методов управления допуском объектов в кластер Kubernetes. Как и многие другие аспекты, рассматриваемые в этой книге, каждый из рассмотренных методов имеет свои плюсы и минусы и требует принятия определенных решений на основе ваших конкретных условий. Управление допуском — это область, в которой необходимы еще более тщательный анализ и глубокие знания, учитывая, насколько важную роль она играет в безопасности кластера и приложений.

Встроенные контроллеры имеют хороший набор возможностей, но вам вряд ли их хватит. Мы ожидаем, что все больше и больше этих функций будут выноситься во внешние контроллеры (не входящие в ядро Kubernetes), которые используют веб-хуки для изменения и проверки запросов. Очень скоро вы можете столкнуться с необходимостью создания собственных веб-хуков (либо с нуля, либо с применением платформы разработки) для поддержки более сложных возможностей. Но, учитывая, что средства управления допуском, такие как Gatekeeper, становятся все более зрелыми, нам кажется, что многие из этих возможностей могут быть реализованы с их помощью.

ГЛАВА 9

Наблюдаемость

Возможность наблюдать за некоторой произвольной программной системой крайне важна. Если вы не можете исследовать состояние своих активных приложений, то вы не в состоянии ими эффективно управлять. Именно это мы имеем в виду под наблюдаемостью (observability): различные механизмы и системы, с помощью которых мы анализируем состояние выполняющегося программного обеспечения, за которое мы отвечаем. Следует признать, что в данном контексте мы отклоняемся от определения наблюдаемости, которое дается в теории управления. Мы решили употребить этот термин просто ввиду его популярности, и чтобы читателям было легче понять, о чем идет речь.

Компоненты, обеспечивающие наблюдаемость, можно разделить на три категории:

- *Журналирование* — агрегация и хранение информации о событиях, записываемой программами.
- *Метрики* — сбор хронологических данных, отображение их на информационных панелях и использование их для отправки оповещений.
- *Трассировка* — захват информации о запросах, которые проходят через несколько разных приложений в кластере.

В этой главе речь пойдет о реализации эффективного механизма наблюдаемости для платформ на основе Kubernetes, который позволит безопасно управлять вашей платформой и размещенными в ней приложениями в реальных условиях. Для начала мы поговорим о журналировании и исследуем средства для агрегирования журнальных записей и передачи их системе ведения журнала вашей компании. Затем мы покажем, как собирать метрики, как их визуализировать и как генерировать на их основе оповещения. В заключение будет рассмотрен процесс отслеживания (трассировки) запросов по мере их перемещения по распределенным системам, это позволит вам лучше понять, что происходит в вашей системе, приложения в которой состоят из отдельных приложений. Давайте же приступим к теме журналирования и рассмотрим распространенные методики, которые успешно применяются в данной области.

Принцип работы журналирования

Этот раздел посвящен механизмам журналирования в платформах на основе Kubernetes. Нас в основном интересует то, как собирать, обрабатывать и направлять журнальные записи из компонентов вашей платформы и развернутых приложений в хранилище.

Давным-давно программное обеспечение, которое мы выполняли в условиях эксплуатации, обычно хранило журнальные записи в файле на диске. Агрегация этих записей (если таковая вообще проводилась) была проще, поскольку у нас было меньше типов приложений и их экземпляров по сравнению с современными системами. В мире контейнеров наши приложения обычно направляют журнальные записи в стандартный вывод или стандартный вывод ошибок, как это делают интерактивные интерфейсы командной строки. Такой подход стал предпочтительным в современном сервис-ориентированном программном обеспечении еще до того, как контейнеры получили широкое распространение. В облачно-ориентированных экосистемах применяется больше типов приложений и экземпляров каждого из них, но в то же время эти задания носят временный характер, и к ним зачастую не подключен диск, куда можно сохранить журнальные записи — отсюда и тенденция к отказу от записи журнала на диск. В результате возникают определенные трудности, связанные со сбором, агрегированием и хранением журнальных записей.

Нередко у одного приложения есть несколько реплик, и у нас может быть много компонентов, которые нужно анализировать. В таких случаях без централизованной агрегации журнальных записей их анализ (просмотр и разбор) становится чрезвычайно утомительным или вообще невозможным. Представьте, что вам нужно проанализировать журнальные записи приложения с *десятками* реплик. Для этого необходима единая точка сбора информации, которая позволяет искать журнальные записи в разных репликах.

При обсуждении механизмов журналирования мы сначала рассмотрим стратегии сбора и передачи журнальных записей из контейнерных приложений в вашей платформе. Это включает в себя записи для управляющей плоскости Kubernetes, служебных компонентов платформы и пользовательских приложений. В данном разделе мы также поговорим о журнале аудита API-сервера и событиях Kubernetes. В заключение мы покажем, как генерировать оповещения в зависимости от журнальных данных, и рассмотрим несколько альтернативных стратегий, которые можно для этого использовать. Мы не станем углубляться в тему хранения журналов, так как у большинства организаций есть своя система хранения, с которой нужно интегрироваться — платформы на основе Kubernetes этим, как правило, не занимаются.

Обработка журнальных записей контейнера

Давайте рассмотрим три метода обработки журнальных записей для контейнерных приложений в платформе на основе Kubernetes:

- *Немедленная отправка* — приложение сразу отправляет журнальные записи в центральный журнал.
- *Прицепной компонент* — для управления журнальными записями приложения служит sidecar-компонент.
- *Отправка с помощью агента на узле* — на каждом узле размещается Pod, которая направляет журнальные записи всех контейнеров этого узла в центральный журнал.

Немедленная отправка

В этом случае приложение должно быть интегрировано с центральным хранилищем журнальных записей. Разработчикам необходимо предусмотреть эту функциональность в своем коде и заниматься ее сопровождением. Если центральное хранилище поменяется, приложение, скорее всего, придется обновить. Поскольку обработка журнальных записей проводится практически повсеместно, куда более разумным решением будет вынести эту возможность за пределы приложения. В большинстве ситуаций немедленная отправка — это не самый лучший вариант, и она редко встречается в реальных окружениях. Ее имеет смысл применять только для устаревших приложений, которые находятся в процессе перехода на платформу, основанную на Kubernetes и уже интегрированную с центральным хранилищем.

Прицепной компонент

В этой модели приложение выполняется в одном контейнере, а журнальные записи оно сохраняет в один или несколько файлов в общем хранилище Pod'а. Другой, sidecar контейнер в том же Pod'е считывает эти записи, обрабатывает их и выполняет одно из двух действий:

1. Направляет их непосредственно в центральное хранилище журнальных записей.
2. Сохраняет их в поток stdout или stderr.

Отправка напрямую в центральное хранилище — основной вариант использования этой модели. Данный подход встречается редко и обычно служит временным решением на случай, если платформа не предлагает систему агрегации журнальных записей.

В ситуациях, когда sidecar контейнер направляет журнальные записи в стандартный вывод или стандартный вывод ошибок, дальнейшим перенаправлением занимается агент узла (подробней об этом в следующем разделе). Этот метод тоже не очень распространен и имеет смысл только в том случае, если выполняемое вами приложение попросту не умеет сохранять журнальные записи в stdout и stderr.

Отправка с помощью агента на узле

В этой модели обработкой журнальных записей занимается приложение, размещенное на каждом узле кластера, оно считывает журнальные файлы всех контейнеров, записанные их средой выполнения, и направляет их в центральное хранилище.

Такой подход мы обычно рекомендуем, и среди всех реализаций эта, несомненно, самая распространенная. Она имеет следующие преимущества:

- Наличие единой точки интеграции между компонентом, направляющим журнальные записи, и центральным хранилищем. Нам не нужны для этого разные sidecar контейнеры и/или приложения.
- Возможен централизованный процесс настройки стандартных средств фильтрации, прикрепления метаданных и отправки записей в несколько хранилищ.

- За ротацию журнальных файлов отвечает kubelet или среда выполнения контейнеров. Если приложение хранит журнальные файлы внутри контейнера, то их ротацией должно заниматься либо оно само, либо sidecar контейнер (если таковой имеется).

Чаще всего для реализации этой модели применяются такие инструменты, как Fluentd (**https:// www.fluentd.org**) и Fluent Bit (**https://fluentbit.io**). Как можно догадаться по названиям, это родственные проекты. Fluentd появился первым, он написан в основном на Ruby и имеет богатую экосистему подключаемых модулей. Проект Fluent Bit стал ответом на необходимость в более легковесном решении для таких окружений, как встраиваемые системы под управлением Linux. Он написан на С и занимает намного меньше места в памяти по сравнению с Fluentd. С другой стороны, для него доступно не так много подключаемых модулей.

Обычно, когда разработчики платформы выбирают средства агрегации и передачи журнальных записей, мы рекомендуем использовать Fluent Bit, если только у Fluentd нет подключаемых модулей с нужными вам возможностями. Если у вас возникает необходимость в подключаемых модулях Fluentd, попробуйте развернуть этот инструмент в качестве общекластерного агрегатора вместе с Fluent Bit. В рамках этой модели Fluent Bit выступает агентом узла, который развертывается в виде DaemonSet. Fluent Bit направляет журнальные записи сервису Fluentd, который работает в кластере как объект `Deployment` или `StatefulSet`. Fluentd выполняет дальнейшую маркировку и направляет журнальные записи в одно или несколько хранилищ, где ими смогут воспользоваться разработчики. Описанный подход проиллюстрирован на рис. 9.1.

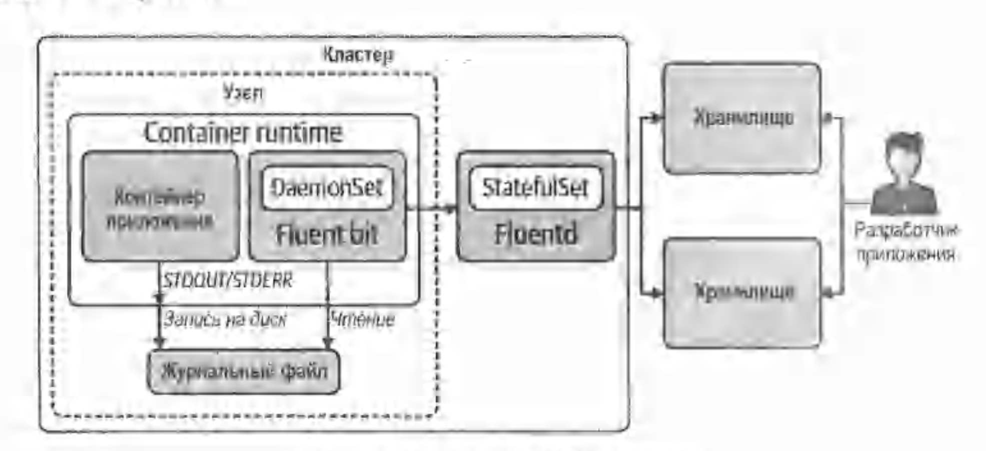


Рис. 9.1. Агрегация журнальных записей из контейнерных приложений в хранилище

Несмотря на то, что мы отдаем явное предпочтение методу с агентом на узле, следует упомянуть потенциальные проблемы, присущие централизованной агрегации журнальных записей. У каждого узла (или целого кластера, если вы применяете общекластерный агрегатор) появляется единая точка отказа. Если агент будет занят каким-то одним приложением, которое активно записывает в журнал, это может

сказаться на сборе журнальных записей всех остальных приложений на том же узле. При наличии общекластерного агрегатора Fluentd, работающего в виде объекта `Deployment`, он будет использовать в качестве буфера слой временного хранилища в своем Pod'е. Если его уничтожат, и он не успеет вывести содержимое буфера, журнальные записи будут утеряны. В связи с этим Fluentd лучше развертывать в виде объекта `StatefulSet`, чтобы журнальные записи не терялись, когда Pod прекращает работу.

Журналы аудита в Kubernetes

Этот раздел посвящен сбору журналов аудита из API-интерфейса Kubernetes. Такие журналы позволяют выяснить, кто выполнил в кластере те или иные действия. Данную возможность стоит включить в реальное окружение, чтобы в случае возникновения каких-то проблем вы смогли определить причины случившегося. Ее включение также может быть продиктовано нормативно-правовыми требованиями.

Журналы аудита включаются и настраиваются с помощью флагов API-сервера. API-сервер позволяет записывать в журнал каждый этап каждого запроса, который ему приходит, включая содержимое запросов и ответов. В реальности вам вряд ли захочется записывать *каждый* запрос. API-сервер принимает множество вызовов, поэтому вам придется хранить большое количество журнальных записей. Вы можете задать правила в политике аудита, чтобы указать, для каких запросов и этапов API-сервер должен сохранять журнальные записи. Если политика аудита отсутствует, в журнал ничего не записывается. Чтобы указать API-серверу, в каком месте файловой системы на узле управляющей плоскости находится политика аудита, используйте флаг `--audit-policy-file`. В листинге 9.1 проиллюстрирована работа нескольких правил, которые позволяют ограничить объем журнальных данных, не отказываясь от важной информации.

Листинг 9.1. Пример политики аудита

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: None ❶
  users: ["system:kube-proxy"]
  verbs: ["watch"]
  resources:
  - group: "" # core
    resources: ["endpoints", "services", "services/status"]
- level: Metadata ❷
  resources:
  - group: ""
    resources: ["secrets", "configmaps"]
  - group: authentication.k8s.io
    resources: ["tokenreviews"]
```

```
  omitStages:
  - "RequestReceived"
- level: Request ❸
  verbs: ["get", "list", "watch"]
  resources:
  - group: ""
  - group: "apps"
  - group: "batch"
  omitStages:
  - "RequestReceived"
- level: RequestResponse ❹
  resources:
  - group: ""
  - group: "apps"
  - group: "batch"
  omitStages:
  - "RequestReceived"
# Уровень по умолчанию для всех остальных запросов
- level: Metadata ❺
  omitStages:
  - "RequestReceived"
```

❶ Уровень аудита `None` означает, что API-сервер не будет записывать в журнал события, соответствующие этому правилу. Поэтому, когда пользователь `system:kube-proxy` запросит подписку на перечисленные ресурсы, это событие не будет записано.

❷ Уровень `Metadata` означает, что в журнал записываются только метаданные запросов. Когда API-сервер принимает какой-либо запрос для перечисленных ресурсов, он сохраняет в журнал сведения о том, какой пользователь сделал запрос того или иного типа для того или иного ресурса, но не само содержимое запроса и ответа. Не попадет в журнал и этап `RequestReceived`. Это означает, что отдельная журнальная запись будет создана не при получении запроса, а когда API-сервер начнет возвращать информацию в рамках длительной подписки. Еще одна запись будет создана после возвращения ответа клиенту или при возникновении серьезной ошибки. Но при получении запроса в журнал ничего не записывается.

❸ На уровне `Request` API-сервер будет записывать в журнал метаданные и содержимое запросов, но *не* содержимое ответов. Поэтому, когда клиент пошлет запрос типа `get`, `list` или `watch`, потенциально объемное тело ответа, содержащее один или несколько объектов, не попадет в журнал.

❹ На уровне `RequestResponse` в журнал записывается бóльшая часть информации: метаданные запроса, а также содержимое запроса и ответа. В этом правиле перечислены те же API-группы, что и в предыдущем, поэтому оно фактически сводится к следующему: если запрос *не* имеет тип `get`, `list` или `watch` и не относится к ресурсу в одной из этих групп, в дополнение ко всему в журнал записывается еще и тело ответа. Это, в сущности, становится уровнем журналирования по умолчанию для перечисленных групп.

❺ Ко всем остальным ресурсам, на которые не распространяются предыдущие правила, будет по умолчанию применено следующее условие: при получении запроса в журнал ничего не записывается, и в дальнейшем журнальные записи будут включать в себя только метаданные запроса, без содержимого запроса и ответа.

Журналы аудита, как и любые другие в вашей системе, нужно направлять в какое-то центральное хранилище. Это можно делать на уровне приложения или с помощью агента на узле, как обсуждалось ранее. Многие из тех принципов и подход, которые мы рассматривали, применимы и здесь.

Если использовать модель с немедленной отправкой, API-сервер можно настроить так, чтобы он направлял журнальные записи непосредственно внутреннему веб-хуку. В таком случае вам нужно указать API-серверу флаг с местоположением конфигурационного файла, который содержит адрес и учетные данные для подключения. Этот файл должен иметь формат kubeconfig. Вам придется потратить некоторое время на то, чтобы подобрать подходящие параметры для буферизации и пакетирования, чтобы все журнальные записи доходили до хранилища. Например, если выбрать максимальное число событий в буфере до их объединения, и оно окажется недостаточным, то буфер переполнится, и события будут утеряны.

Если отправка осуществляется с помощью агента на узле, то вы можете сделать так, чтобы API-сервер записывал журнальные файлы на файловую систему узла управляющей плоскости. API-серверу можно предоставить флаги для задания пути к файлу, наибольшего периода хранения, максимального числа файлов и предельного размера журнального файла. В этом случае журнальные записи можно агрегировать и передавать с помощью таких инструментов, как Fluent Bit и Fluentd. Это, скорее всего, будет хорошим решением для тех, кто уже применяет данные инструменты для управления журнальными записями в рамках модели на основе агента узла, рассмотренной ранее.

События Kubernetes

События в Kubernetes являются стандартными ресурсами типа Event. С их помощью компоненты платформы предоставляют информацию о том, что происходит с разными объектами в рамках API-интерфейса Kubernetes. Это что-то вроде журнала платформы. Но, в отличие от обычных журнальных записей, события обычно не записываются в отдельное хранилище. Они хранятся в etcd не дольше определенного времени, которое по умолчанию составляет один час. Их чаще всего применяют администраторы и пользователи платформы, которые хотят собирать информацию о действиях, производимых с объектами. В листинге 9.2 показаны События, перечисленные в описании только что созданного Pod'а.

Листинг 9.2. События в описании Pod'а

```
$ kubectl describe pod nginx-6db489d4b7-q8ppw
Name:         nginx-6db489d4b7-q8ppw
Namespace:    default
...
```

```
Events:
  Type    Reason     Age        From
  ----    ------     ----       ----
    Message
    -------
  Normal  Scheduled  <unknown>  default-scheduler
    Successfully assigned default/nginx-6db489d4b7-q8ppw
  Normal  Pulling    34s        kubelet, ip-10-0-0-229.us-east-2.compute.internal
    Pulling image "nginx"
  Normal  Pulled     30s        kubelet, ip-10-0-0-229.us-east-2.compute.internal
    Successfully pulled image "nginx"
  Normal  Created    30s        kubelet, ip-10-0-0-229.us-east-2.compute.internal
    Created container nginx
  Normal  Started    30s        kubelet, ip-10-0-0-229.us-east-2.compute.internal
    Started container nginx
```

Эти же События можно получить напрямую, как показано в листинге 9.3. В данном случае в дополнение к Событиям, которые мы видели в описании Pod'а, выводятся События для ресурсов `ReplicaSet` и `Deployment`.

Листинг 9.3. События для пространства имен, полученные напрямую

```
$ kubectl get events -n default
LAST SEEN   TYPE     REASON            OBJECT
    MESSAGE
2m5s        Normal   Scheduled             pod/nginx-6db489d4b7-q8ppw
    Successfully assigned default/nginx-6db489d4b7-q8ppw
2m5s        Normal   Pulling               pod/nginx-6db489d4b7-q8ppw
    Pulling image "nginx"
2m1s        Normal   Pulled                pod/nginx-6db489d4b7-q8ppw
    Successfully pulled image "nginx"
2m1s        Normal   Created               pod/nginx-6db489d4b7-q8ppw
    Created container nginx
2m1s        Normal   Started               pod/nginx-6db489d4b7-q8ppw
    Started container nginx
2m6s        Normal   SuccessfulCreate      replicaset/nginx-6db489d4b7
    Created pod: nginx-6db489d4b7-q8ppw
2m6s        Normal   ScalingReplicaSet    deployment/nginx
    Scaled up replica set nginx-6db489d4b7 to 1
```

Поскольку События доступны через API-интерфейс Kubernetes, их отслеживание и обработку вполне можно автоматизировать. Но, как показывает наш опыт, этим занимаются немногие. Вы также можете воспользоваться средством экспорта событий, чтобы просматривать их в виде метрик. В разделе "Prometheus" данной главы обсуждаются средства экспорта Prometheus.

Генерация оповещений на основе журнальных записей

Журнальные записи приложений содержат важную информацию о поведении вашего программного обеспечения. Они особенно важны при возникновении неожиданных сбоев, которые нужно расследовать. Вы можете обнаружить цепочку событий, которая привела к проблемам. Если вам захочется настроить оповещения, генерируемые в ответ на События в журнальных записях, попробуйте воспользоваться вместо этого метриками. Если открыть доступ к метрикам, представляющим это поведение, для них можно реализовать правила создания оповещений. Журнальные сообщения больше подвержены изменениям, поэтому оповещения на их основе получаются менее надежными. Любое изменение текста в журнальном сообщении может непреднамеренно нарушить работу системы оповещений, которая его использует.

Последствия для безопасности

Не забудьте выяснить, какой доступ имеют пользователи к различным журнальным записям, агрегированным в вашем центральном хранилище. Вам, наверное, не хочется, чтобы журналы аудита вашего API-сервера были доступны всем подряд. У вас могут быть системы с конфиденциальной информацией, предназначенной только для привилегированных пользователей, что может повлиять на маркирование журнальных записей или даже потребовать использования нескольких хранилищ, что повлечет за собой изменение параметров переадресации.

Итак, мы обсудили различные механизмы, участвующие в управлении журнальными записями вашей платформы и развернутых на ней приложений. Теперь перейдем к метрикам и оповещениям.

Метрики

Метрики и сервисы оповещения имеют важнейшее значение для удобства использования платформы. С помощью метрик мы можем нанести собранные данные на временную шкалу и увидеть отклонения, свидетельствующие о нежелательном или непредвиденном поведении. Они помогают нам понять, что происходит с нашими приложениями, ведут ли они себя так, как задумывалось, и каким образом мы можем исправить проблемы или улучшить управление своими приложениями. И, что крайне важно, метрики дают нам полезные показатели, на основе которых можно генерировать оповещения. Уведомления об уже случившихся или, еще лучше, предупреждения о предстоящих сбоях позволяют предотвратить и/или минимизировать ошибки и вынужденные простои.

В этом разделе речь пойдет о предоставлении метрик и организации оповещений в виде сервиса платформы с использованием Prometheus. Эта область полна нюансов, поэтому будет полезно взять в качестве примера конкретный стек программного обеспечения. Сказанное вовсе не означает, что вы не можете или не должны применять другие решения. Существует много ситуаций, в которых Prometheus может

оказаться *неоптимальным* выбором. С другой стороны, Prometheus предлагает замечательный подход к решению рассматриваемых здесь задач. На каком бы инструменте вы ни остановились, модель Prometheus послужит понятной эталонной реализацией, которая даст вам представление о том, как подходить к этим вопросам.

Для начала мы проведем общий обзор проекта Prometheus и поговорим о том, что он собой представляет, как он собирает метрики и какие у него возможности. Затем будут рассмотрены различные вопросы общего характера, такие как долгосрочное хранение и сценарий с принудительной отправкой. Далее мы обсудим генерацию и сбор пользовательских метрик, а также организацию процесса сбора метрик в рамках всей инфраструктуры, в том числе и федеративной (многокластерной). Также будет затронута тема оповещений и использования метрик для оценки потребляемых ресурсов и их стоимости. В конце мы разберем каждый из разнообразных компонентов в стеке Prometheus и проиллюстрируем их совместную работу.

Prometheus

Prometheus — это средство управления метриками с открытым исходным кодом, получившее широкое распространение в платформах на основе Kubernetes. Метрики в этом формате предоставляют компоненты плоскости управления, и практически любой промышленный кластер за счет средств экспорта Prometheus получает метрики из таких источников, как внутренние узлы. В связи с этим многие корпоративные системы, такие как Datadog, New Relic и VMware Tanzu Observability, поддерживают потребление метрик Prometheus.

Метрики Prometheus — это просто стандартный формат хронологических данных, который в реальности может применяться в любой системе. Prometheus использует активную модель, в рамках которой метрики собираются из источников. Таким образом, приложения и инфраструктурные компоненты обычно не занимаются *отправкой* метрик самостоятельно, они просто открывают к ним доступ в виде конечной точки, а Prometheus уже их оттуда достает. Такой подход избавляет приложение от необходимости знать что-либо о системе метрик, не считая формата, в котором представлены данные.

Активная модель сбора метрик, возможность обрабатывать большие объемы информации, использование меток в ее модели данных и язык PromQL (Prometheus Query Language) делают Prometheus отличным средством управления метриками для динамичных облачно-ориентированных окружений. Мы можем легко добавлять и отслеживать новые приложения, предоставлять доступ к метрикам приложения или системы, добавлять конфигурации сбора метрик в сервер Prometheus и с помощью PromQL преобразовывать исходные данные в полезную информацию и оповещения. Вот лишь некоторые из основных причин популярности Prometheus в экосистеме Kubernetes.

Prometheus обладает несколькими важнейшими возможностями по работе с метриками:

- собирает метрики из источников с помощью своей модели извлечения данных;
- хранит метрики в базе данных с поддержкой временных рядов;

- отправляет оповещения (обычно диспетчеру Alertmanager) на основе соответствующих правил, о чем мы поговорим позже в этой главе;
- предоставляет другим компонентам HTTP-интерфейс для доступа к метрикам, хранящимся в Prometheus;
- предоставляет информационную панель, с помощью которой можно быстро составлять и выполнять запросы к метрикам и получать различные сведения о состоянии.

Большинство разработчиков изначально применяют Prometheus для сбора метрик в сочетании с Grafana для визуализации. Однако для небольших коллективов программистов организованное использование такой конфигурации в реальном окружении может стать проблемой. Вам придется решить вопросы долгосрочного хранения своих метрик, масштабирования Prometheus с ростом объемов данных и организации федеративного управления вашими системами метрик. Ни одна из указанных проблем не является тривиальной, и их решение со временем потребует существенных усилий. Поэтому, если обслуживание стека метрик становится громоздким по мере увеличения масштаба системы, вы можете перейти на один из коммерческих продуктов, не меняя при этом тип метрик, которые у вас приняты.

Долгосрочное хранение

Необходимо отметить, что проект Prometheus не предназначен для долгосрочного хранения метрик. Вместо этого он дает возможность осуществлять запись в удаленные конечные точки, и для такого рода интеграции можно воспользоваться рядом решений (**https://oreil.ly/wcaVl**). При предоставлении механизма работы с метриками в рамках платформы приложений возникают вопросы, связанные с хранением данных. Будет ли доступно долгосрочное хранение только в условиях эксплуатации? Если да, то как долго метрики могут храниться на уровне Prometheus в остальных окружениях? Каким образом пользователи смогут получать доступ к метрикам в долгосрочном хранилище? Такие проекты, как Thanos (**https://thanos.io**) и Cortex (**https://cortexmetrics.io**), предлагают инструментарий для решения этих задач. Но следите за тем, как ваши пользователи применяют эти системы, и дайте им знать о том, какие политики хранения данных они могут ожидать.

Пассивная модель сбора метрик

Активная модель сбора метрик подходит не для всякого приложения. В таких случаях можно использовать Prometheus Pushgateway (**https://github.com/prometheus/pushgateway**). Например, приложение с пакетной обработкой, которое останавливается по завершении работы, дает серверу Prometheus шанс собрать все метрики, пока они не исчезли. В таких ситуациях приложение может передать свои метрики инструменту Pushgateway, который в свою очередь сделает их доступными для сервера Prometheus. Поэтому, если ваша платформа будет поддерживать этот сценарий, вам, вероятно, нужно развернуть Pushgateway как один из компонентов для работы с метриками и сделать информацию доступной для приложений, чтобы они

могли ею пользоваться. Приложения должны знать, где именно в кластере находится Pushgateway и как работать с его REST-подобным HTTP-интерфейсом. На рис. 9.2 показано, как временное приложение задействует клиентскую библиотеку Prometheus, которая поддерживает отправку метрик в Pushgateway. В конечном итоге, эти метрики собираются сервером Prometheus.



Рис. 9.2. Pushgateway для временных приложений

Пользовательские метрики

Доступ к метрикам Prometheus можно открывать стандартными средствами. Многие приложения, разработанные специально для платформ на основе Kubernetes, именно так и поступают. Существует также несколько официально поддерживаемых клиентских библиотек (**https://oreil.ly/t9SLv**) и ряд аналогов, разрабатываемых сообществом. Благодаря им разработчики приложений, скорее всего, смогут с легкостью открывать доступ к пользовательским метрикам, готовым для сбора системой Prometheus. Это подробно обсуждается в *главе 14*.

Если же приложение или система не имеет встроенной поддержки метрик Prometheus, можно воспользоваться *средствами экспорта*. Они извлекают данные из приложения или системы и затем предоставляют их в формате Prometheus. Распространенный пример — инструмент Node Exporter. Он собирает метрики об оборудовании и операционной системе и делает их доступными в том виде, в котором Prometheus сможет их собрать. Существуют средства экспорта, поддерживаемые сообществом (**https://oreil.ly/JO8sO**) и предназначенные для широкого спектра популярных инструментов. Некоторые из них могут вам пригодиться.

После развертывания приложения, предоставляющего пользовательские метрики, его нужно добавить в конфигурацию сервера Prometheus. Это обычно делается с помощью пользовательского ресурса ServiceMonitor, с которым работает Prometheus Operator. О Prometheus Operator мы подробно поговорим далее в разделе "Компоненты для работы с метриками", а пока достаточно сказать, что с помощью пользовательского ресурса Kubernetes можно сделать так, чтобы этот инструмент автоматически обнаруживал Сервисы в определенном пространстве имен и с заданными метками.

Если подытожить, то по возможности инструментируйте свое программное обеспечение стандартными средствами. Если встроенные инструменты не подходят, разработайте собственные или воспользуйтесь готовыми средствами экспорта. Собирайте доступные метрики с помощью удобных механизмов автоматического обнаружения, чтобы сделать свои системы наблюдаемыми.



Применение меток в модели данных Prometheus открывает большие возможности, которые требуют к себе ответственного отношения. Они могут принести не только пользу, но и вред. Чрезмерное использование меток может повысить потребление ресурсов вашими серверами Prometheus до неприемлемого уровня. Вы должны осознать последствия, которыми чревато наличие мощной системы метрик, и ознакомиться с руководством по инструментированию (**https://oreil.ly/RAskV**) в документации Prometheus.

Организация метрик и федеративные системы

Обработка метрик может быть чрезвычайно ресурсоемкой, поэтому распределение данной вычислительной нагрузки может помочь удержать потребление ресурсов серверами Prometheus под контролем. Например, один сервер Prometheus может собирать метрики платформы, а другой (или другие) — метрики приложений или узлов. Это особенно касается крупных кластеров с большим количеством источников для сбора данных и большими объемами метрик, которые нужно обрабатывать.

С другой стороны, собранные данные будут доступны в разных местах, что приведет к фрагментации. Решить эту проблему можно за счет объединения в федерацию. В сущности, это означает объединение данных в централизованной системе. В нашем контексте это выглядит как сбор важных метрик из различных серверов Prometheus и сохранение их на центральном сервере Prometheus. Для этого применяется та же активная модель, что и для сбора метрик приложений. Одним из источников метрик для сервера Prometheus может служить другой такой сервер.

Такого результата можно добиться в пределах одного и/или нескольких кластеров Kubernetes. В результате получается очень гибкая модель в том смысле, что вы можете организовывать и объединять свои системы метрик в соответствии с тем, как вы управляете своими кластерами Kubernetes. Это включает в себя объединение в многослойную или многоуровневую федерацию. На рис. 9.3 показан пример глобального сервера Prometheus, который собирает метрики из серверов Prometheus, размещенных в разных центрах обработки данных, а те в свою очередь занимаются сбором метрик из источников в пределах своего кластера.

Несмотря на мощь и гибкость федерации Prometheus, она может быть сложной и обременительной в управлении. Относительно недавно появился проект с открытым исходным кодом Thanos (**https://thanos.io**), который дополняет Prometheus возможностями, подобными объединению в федерацию. Это довольно практичное средство для сбора метрик из всех серверов Prometheus. Оно поддерживается системой Prometheus Operator и может развертываться поверх уже установленных копий Prometheus. Еще один проект, подающий надежды в этой области, — Cortex (**https://cortexmetrics.io**). И Thanos, и Cortex развиваются в рамках инкубатора CNCF.

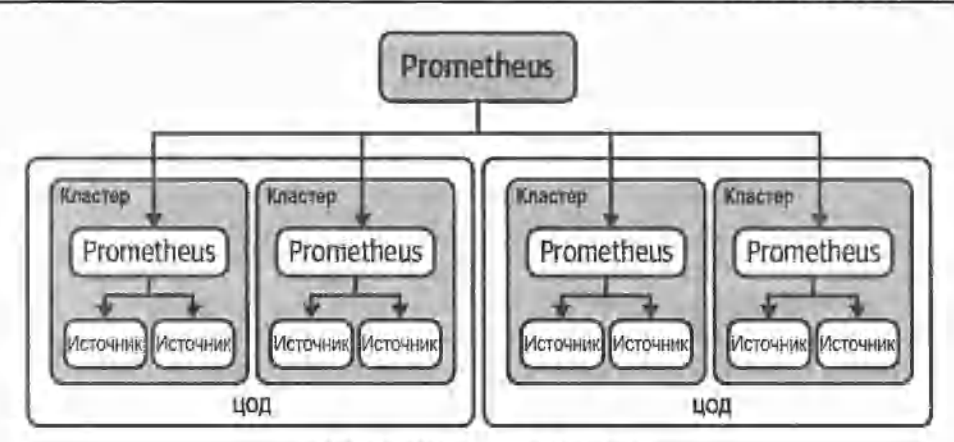


Рис. 9.3. Федерация Prometheus

Тщательно планируйте организацию и объединение в федерацию своих серверов Prometheus, чтобы они справлялись с масштабированием и расширением вашей платформы по мере того, как у нее появляется все больше пользователей. Как следует продумайте модель потребления ресурсов. Не допускайте того, чтобы пользователям приходилось открывать множество разных информационных панелей для доступа к метрикам своих приложений.

Оповещения

Для генерации оповещений на основе метрик Prometheus использует специальные правила. При срабатывании оповещения обычно отправляются сконфигурированному экземпляру Alertmanager. Развертывание Alertmanager и настройка Prometheus для оповещений существенно упрощаются при наличии Prometheus Operator. Alertmanager обработает оповещение и уведомит ваших инженеров о проблемах с помощью систем обмена сообщениями. На рис. 9.4 проиллюстрировано применение отдельных серверов Prometheus для плоскости управления и развернутых в ней приложений. В обоих случаях обработка оповещений и уведомление получателей выполняется с помощью Alertmanager.

В целом старайтесь не создавать *слишком много* оповещений, так как это может утомить ваших дежурных инженеров, а переизбыток ложных срабатываний способен затмить собой по-настоящему важные события. Поэтому потратьте время на то, чтобы сделать свои оповещения полезными. Добавьте информативные описания в аннотации своих оповещений, чтобы инженеры, которые их получают, имели контекст для понимания ситуации. Попробуйте включать в runbook или другие документы ссылки, которые могут помочь в разрешении проблемы.

Помимо оповещений для платформы подумайте над тем, как предоставить механизм оповещений вашим пользователям, чтобы они могли применять его для метрик в своих приложениях. Доступ к этому механизму подразумевает возможность добавления правил оповещения в Prometheus, о чем мы поговорим подробней далее

в разделе "Компоненты для работы с метриками". Это также означает, что ваши пользователи смогут настраивать механизмы оповещений с помощью Alertmanager так, чтобы разработчики приложений получали уведомления в соответствии с правилами, которые они установили.

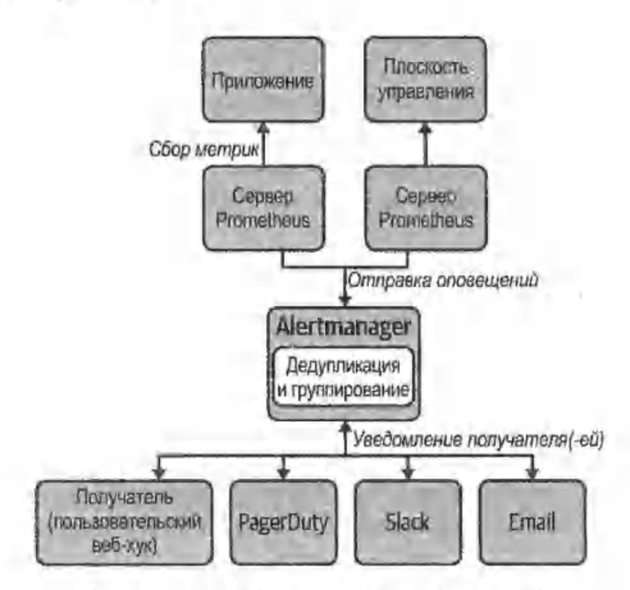


Рис. 9.4. Компоненты механизма оповещений

Контроль исправности оповещений

Есть один тип оповещений, заслуживающий отдельного внимания ввиду его особого значения и возможности применения в практически любых системах. Что произойдет, если ваши системы метрик и оповещений выйдут из строя? Как получить оповещение о *таком* событии? На этот случай вам нужно подготовить оповещение, которое будет периодически срабатывать в штатном режиме работы. Если оно перестанет приходить, должно быть сгенерировано приоритетное оповещение, которое даст вам знать, что ваши системы метрик и/или оповещений не работают. У инструмента PagerDuty (**https://oreil.ly/zDJJE**) есть механизм под названием Dead Man's Snitch[1], предоставляющий эту возможность. В качестве альтернативы для устанавливаемой вами системы можно предусмотреть собственное решение с оповещениями на основе веб-хука. Какой бы ни была ваша реализация, позаботьтесь о том, чтобы при поломках системы оповещений вы получали срочные уведомления.

Потребляемые ресурсы и их стоимость

В этом разделе речь пойдет об описании потребления ресурсов подразделениями организации или их приложениями, а также о финансовых расходах, которые с

[1] Непереводимая игра слов. — *Пер.*

этим связаны. Это идеальный пример количественной оценки метрик, на основе которого можно предпринимать какие-то действия.

Kubernetes предлагает возможность динамического управления вычислительной инфраструктурой, используемой разработчиками приложений. При неосторожном обращении с этим механизмом ваш кластер может стать слишком громоздким, что выльется в неоптимальный расход ресурсов. Организациям крайне выгодно автоматизировать процесс развертывания инфраструктуры и приложений, чтобы сделать его более эффективным. Однако подобная автоматизация может оказаться расточительной. В связи с этим во многих организациях группы сотрудников и целые бизнес-направления ответственны за потребляемые ими ресурсы и связанные с этим финансовые расходы.

Чтобы собирать нужные метрики, наши приложения должны иметь какие-то информативные имена или идентификаторы вроде "team" или "owner". Советуем выработать для этого стандартную систему и применять средства управления допуском, чтобы обеспечить использование таких меток во всех Pod'ах, которые развертываются пользователями платформы. Встречаются и другие полезные методы идентификации приложений (например, по пространству имен), но метки дают наибольшую гибкость.

Существуют два основных подхода к реализации учета ресурсов:

- *На основе запросов* — это ресурсы, которые команда резервирует с помощью запросов, определяемых для каждого контейнера в Pod'е.
- *На основе фактического потребления* — учитываются те ресурсы, которые команда запрашивает *или* в действительности потребляет (в зависимости от того, какой из этих показателей выше).

Учет ресурсов на основе запросов

Этот метод учитывает запросы агрегированных ресурсов, определяемые приложением. Например, если объект Deployment запрашивает по одному ядру процессора для каждой своей реплики, которых у него насчитывается 10, то считается, что на протяжении своей работы он задействовал 10 ядер за единицу времени. Если приложение выйдет за рамки своих запросов и начнет использовать 1,5 ядер для каждой реплики, оно получит эти ресурсы "безвозмездно". Пять ядер, потребленные в дополнение к запрошенным ресурсам, *не* будут приписаны приложению. Преимущество такого подхода заключается в том, что он учитывает, какие ресурсы планировщик может выделить узлам в кластере. С точки зрения планировщика, запрошенные ресурсы становятся зарезервированными для узла. Если узел обладает лишними ресурсами, которые простаивают без дела, и одно из его приложений испытывает всплеск активности, оно получает эти ресурсы "даром". Сплошная линия на рис. 9.5 обозначает ресурсы процессора, приписываемые приложению при реализации этого метода. Потребление, выходящее за рамки запросов, *не* учитывается.

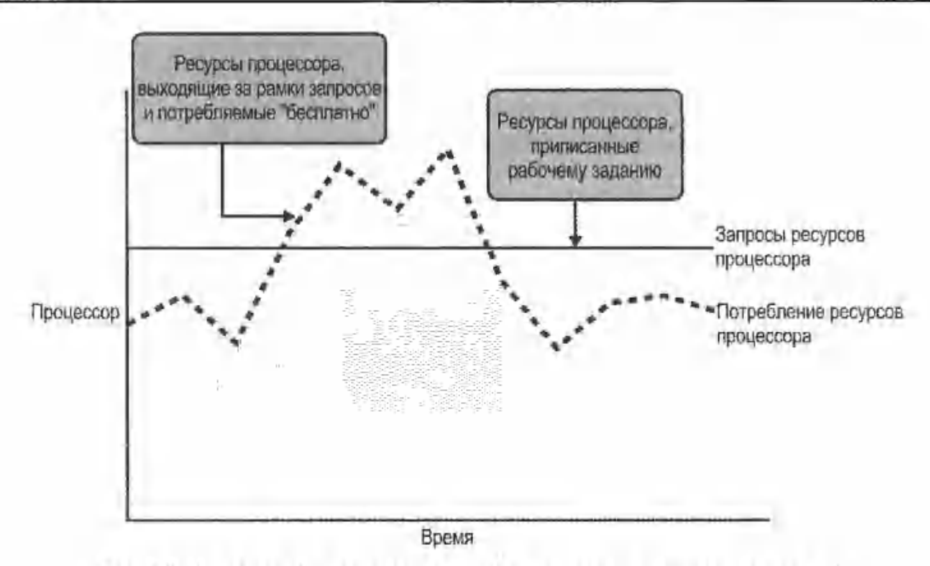


Рис. 9.5. Учет потребления на основе запросов вычислительных ресурсов

Учет ресурсов на основе фактического потребления

В этой модели приложению приписываются ресурсы, которое оно запросило *или* фактически потребило, смотря какой показатель выше. Таким образом, если приложение постоянно и стабильно выходит за рамки запрошенных ресурсов, ему будет приписано то, что оно на самом деле потребило. Если сделать объем запрашиваемых ресурсов небольшим, это уберет потенциальный стимул к поиску лазеек в системе учета ресурсов. С другой стороны, повышается вероятность конкуренции за ресурсы на узлах со слишком большим числом приложений.

Сплошная линия на рис. 9.6 обозначает ресурсы процессора, приписанные приложению при реализации метода, основанного на фактическом потреблении. В данном случае это потребление, выходящее за рамки запрошенных ресурсов.

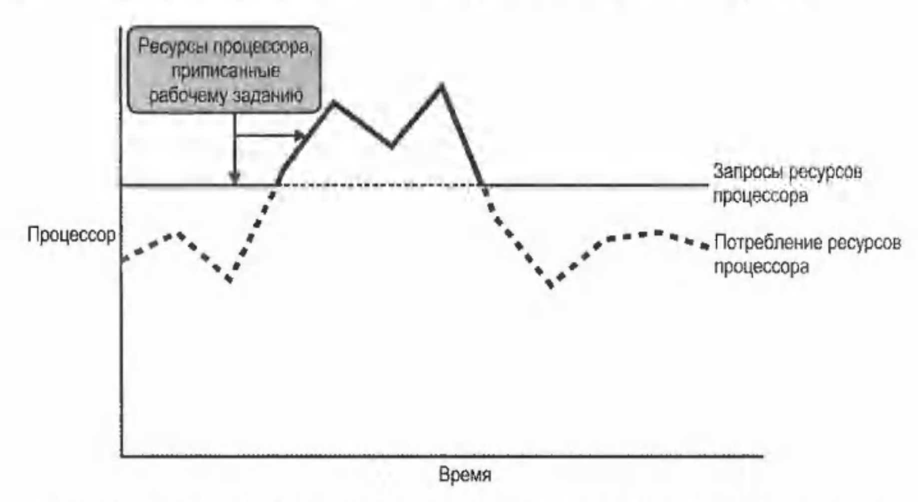


Рис. 9.6. Учет потребления вычислительных ресурсов, выходящих за рамки запросов

В разделе "Компоненты для работы с метриками" данной главы мы обсудим kube-state-metrics, сервис платформы, предоставляющий доступ к метрикам, относящимся к ресурсам Kubernetes. При использовании этого сервиса становятся доступными следующие метрики для запросов ресурсов:

- Процессор — `kube_pod_container_resource_requests`.
- Память — `kube_pod_container_resource_requests_memory_bytes`.

Информацию о потреблении ресурсов можно получить с помощью следующих метрик:

- Процессор — `container_cpu_usage_seconds_total`.
- Память — `container_memory_usage_bytes`.

Напоследок следует сказать, что вы должны решить, что будет учитываться при потреблении ресурсов приложением: процессор или память. Для этого посчитайте, какую долю от общего объема ресурсов кластера (включая память и процессор) потребило приложение, и затем выберите больший показатель, так как кластер, у которого закончился хотя бы один из этих ресурсов, не может обслуживать дополнительные приложения. Например, если приложение использует 1 % от ресурсов процессора и занимает 3 % памяти кластера, то его потребление фактически равно 3 %, поскольку в кластере, у которого закончилась память, больше нельзя размещать новые приложения. В результате вы сможете понять, следует ли использовать другие виды узлов, которые лучше подходят для развернутых на них приложениях (см. раздел "Инфраструктура" *главы 2*).

Учет финансовых расходов

Организовав учет ресурсов, мы можем взяться за финансовые расходы, так как у нас есть метрики, по которым их можно посчитать. Расходы на серверы обычно вычислить довольно просто, если пользоваться услугами общедоступных облачных провайдеров. Самостоятельная покупка оборудования может немного усложнить расчеты, но так или иначе вам нужно получить два показателя:

- расходы на процессоры за единицу времени;
- расходы на память за единицу времени.

Примените их к объему израсходованных ресурсов, который вы определили, и у вас получится модель тарификации пользователей вашей платформы.

Сеть и хранение данных

Итак, мы рассмотрели учет потребления ресурсов и сопутствующих финансовых расходов в контексте вычислительной инфраструктуры, которую используют приложения. Это покрывает бóльшую часть сценариев, которые встречаются нам в реальных условиях. Однако существуют приложения, потребляющие значительные объемы сетевого трафика и места на диске. Подобная инфраструктура может играть существенную роль в реальной стоимости выполнения некоторых приложений, и в таких случаях ее нужно учитывать. Это делается примерно тем же способом, что и

прежде: мы собираем нужные нам метрики и принимаем решение о том, за какие ресурсы взимать плату: зарезервированные, потребленные или за сочетание тех и других. Выбор способа сбора метрики зависит от систем, применяемых в данной инфраструктуре.

Мы обсудили принцип работы Prometheus и общие темы, в которых вы должны ориентироваться, прежде чем приступать к подробному рассмотрению развертываемых компонентов, которые обычно используются в механизме управления метриками Prometheus. Далее мы проведем их общий обзор.

Компоненты для работы с метриками

В этом разделе мы исследуем компоненты широко применяемой модели развертывания и администрирования системы метрик. Мы также обсудим доступные вам средства управления и то, как все это складывается в единую картину.

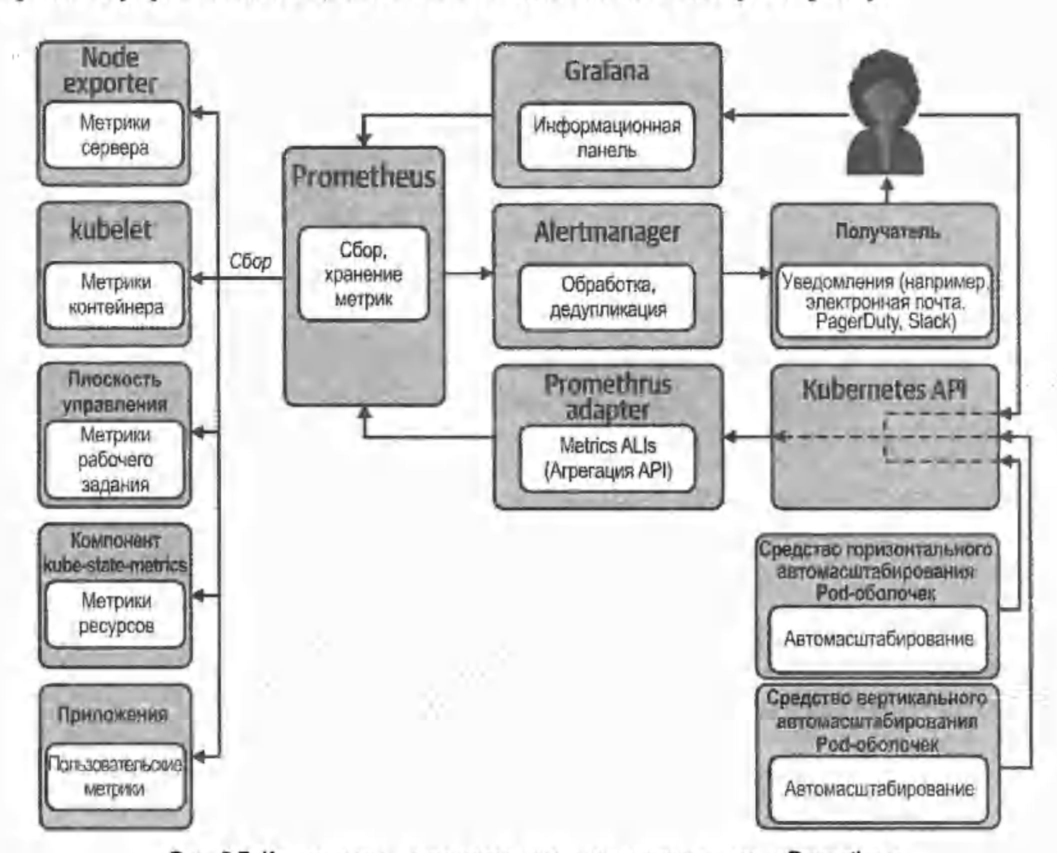


Рис. 9.7. Компоненты, часто используемые в системе метрик Prometheus

На рис. 9.7 проиллюстрировано распространенное сочетание компонентов в системе метрик Prometheus. В него не входит инструмент Prometheus Operator, так как он предназначен для развертывания и администрирования этой системы и не является ее частью. Мы показали на этом рисунке некоторые компоненты автоматического

масштабирования, чтобы продемонстрировать роль Prometheus Adapter, хотя сама тема автоматического масштабирования здесь не рассматривается (подробней о ней можно почитать в *главе 13*).

Prometheus Operator

Prometheus Operator (**https://oreil.ly/k1lMx**) — это оператор Kubernetes, который помогает развертывать и администрировать различные компоненты системы метрик Kubernetes как для самой платформы, так и для размещенных в ней приложений. Подробней об операторах Kubernetes можно почитать в разделе "Шаблон проектирования „оператор"" *главы 11*. Prometheus Operator применяет несколько пользовательских ресурсов, представляющих серверы Prometheus: экземпляры AlertManager, описывающие цели, из которых нужно извлекать метрики, и правила для записи этих метрик и генерации оповещений на их основе. Это существенно облегчает процесс развертывания и конфигурации серверов Prometheus на вашей платформе.

Пользовательские ресурсы сильно помогают разработчикам платформы, но в то же время могут предоставить крайне важный интерфейс для ее пользователей. Если пользователю нужен выделенный сервер Prometheus, он может его получить, применив ресурс Prometheus к своему пространству имен. Если ему нужно добавить в существующий сервер Prometheus правила о генерации оповещений, он может воспользоваться для этого ресурсом PrometheusRule.

Помочь в работе с Prometheus Operator может замечательный родственный проект kube-prometheus (**https://oreil.ly/DITxj**). Он предоставляет набор манифестов для полноценной системы метрик. В его состав входят конфигурации информационных панелей Grafana, позволяющие легко визуализировать данные без дополнительной настройки, что крайне удобно. Но относитесь к этому проекту как к отправной точке для знакомства с системой с последующей ее адаптацией к вашим требованиям, чтобы после ее развертывания в реальном окружении вы могли быть уверены в том, что она имеет все необходимые метрики и оповещения.

Оставшаяся часть этого раздела посвящена компонентам, которые вы получаете при развертывании kube-prometheus, чтобы вы имели о них четкое представление и могли их адаптировать под свои нужды.

Серверы Prometheus

Развернув в своих кластерах Prometheus Operator, вы сможете создавать пользовательские ресурсы Prometheus, на основе которых оператор будет создавать новые объекты `StatefulSet` для серверов Prometheus. Манифест для ресурса Prometheus показан в листинге 9.4.

Листинг 9.4. Пример манифеста Prometheus

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
```

```
metadata:
  name: platform
  namespace: platform-monitoring
  labels:
    monitor: platform
    owner: platform-engineering
spec:
  alerting: ❶
    alertmanagers:
    - name: alertmanager-main
      namespace: platform-monitoring
      port: web
  image: quay.io/prometheus/prometheus:v2.20.0 ❷
  nodeSelector:
    kubernetes.io/os: linux
  replicas: 2
  resources:
    requests:
      memory: 400Mi
  ruleSelector: ❸
    matchLabels:
      monitor: platform
      role: alert-rules
  securityContext:
    fsGroup: 2000
    runAsNonRoot: true
    runAsUser: 1000
serviceAccountName: platform-prometheus
version: v2.20.0
serviceMonitorSelector: ❹
  matchLabels:
    monitor: platform
```

❶ Информирует конфигурацию Prometheus о том, куда посылать оповещения.

❷ Образ контейнера, который нужно использовать для Prometheus.

❸ Говорит системе Prometheus Operator о том, какие правила PrometheusRule нужно назначить для данного сервера Prometheus. Применены будут все правила с метками, показанные здесь.

❹ Выполняет ту же функцию для ServiceMonitor, что и `ruleSelector` для PrometheusRule. Все ресурсы ServiceMonitor с этой меткой будут задействованы в конфигурации сбора метрик этого сервера Prometheus.

Пользовательский ресурс Prometheus позволяет администраторам платформы легко развертывать серверы Prometheus для сбора метрик. Как уже упоминалось в разделе "Организация метрик и федеративные системы" данной главы, иногда полезно распределить процессы сбора и обработки метрик между несколькими экземплярами

Prometheus в пределах заданного кластера. Это можно сделать за счет создания дополнительных серверов Prometheus с помощью пользовательского ресурса Kubernetes.

Иногда возможность создавать серверы Prometheus с помощью Prometheus Operator имеет смысл сделать доступной и для пользователей платформы. Приложения разработчиков могут генерировать большие объемы метрик, которые будут чрезмерно нагружать существующие серверы. К тому же, вы можете захотеть причислить сбор и обработку метрик приложений к ресурсам их разработчиков, поэтому наличие отдельного сервера Prometheus в их пространстве имен может быть полезным. Не всяким разработчикам придется по душе подход, согласно которому они сами должны развертывать и администрировать свои ресурсы Prometheus. Им может потребоваться более высокоуровневая абстракция. Тем не менее, этот вариант заслуживает внимания. Если вы выберете данную модель, не забывайте о том, что она усложнит создание информационных панелей, генерацию оповещений на основе собранных метрик, объединение кластеров в федерацию и долгосрочное хранение.

Развернуть серверы Prometheus — это одно, а их постоянное администрирование и настройка — совсем другое. Для этого Prometheus Operator предлагает другие пользовательские ресурсы, самый распространенный из них — ServiceMonitor. В ответ на его создание Prometheus Operator обновляет конфигурацию сбора метрик для соответствующего сервера Prometheus. В листинге 9.5 показан объект `ServiceMonitor`, создающий конфигурацию, с помощью которой Prometheus будет собирать метрики из API-сервера Kubernetes.

Листинг 9.5. Пример манифеста для ресурса `ServiceMonitor`

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    k8s-app: apiserver
    monitor: platform ❶
  name: kube-apiserver
  namespace: platform-monitoring
spec:
  endpoints: ❷
  - bearerTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
    interval: 30s
    port: https
    scheme: https
    tlsConfig:
      caFile: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      serverName: kubernetes
  jobLabel: component ❸
  namespaceSelector: ❹
    matchNames:
    - default
```

```
  selector: ❺
    matchLabels:
      component: apiserver
      provider: kubernetes
```

❶ Это метка, на которую ссылается `serviceMonitorSelector` в манифесте `Prometheus` из листинга 9.1.

❷ В разделе `endpoints` описывается порт, который нужно использовать, и то, как подключаться к серверам, из которых Prometheus будет извлекать метрики. Согласно конфигурации в этом примере, Prometheus должен подключаться по HTTPS, предоставляя сертификат и имя сервера для проверки подлинности конечной точки соединения.

❸ В терминологии Prometheus "job" (задание) означает коллекцию экземпляров сервиса. Например, отдельный API-сервер является "экземпляром", а все API-серверы в кластере составляют "job". Это поле описывает метку с именем, которое нужно назначить для задания в Prometheus. В данном случае этим заданием будет `apiserver`.

❹ Благодаря полю `namespaceSelector` Prometheus знает, в каком пространстве имен нужно искать Сервисы для сбора метрик из этого источника.

❺ Поле `selector` делает возможным обнаружение Сервисов Kubernetes по их меткам. Иными словами, любой Сервис (в пространстве имен по умолчанию) с заданными метками будет задействован для поиска источников метрик.

Конфигурацией сбора метрик на сервере Prometheus можно также управлять с помощью ресурсов `PodMonitor`, предназначенных для групп мониторинга Pod'ов (а не Сервисов, как в случае с `ServiceMonitor`), и `Probe` для мониторинга объектов `Ingress` или статических источников.

Объект `PrometheusRule` говорит оператору о том, что для Prometheus нужно сгенерировать файл с правилами для записи метрик и создания оповещений на их основе. В листинге 9.6 приведен пример манифеста `PrometheusRule` с двумя правилами: одним — для записи, а другим — для оповещений. Эти правила будут сохранены в `ConfigMap` и подключены к Pod'у сервера Prometheus.

Листинг 9.6. Пример манифеста для ресурса `PrometheusRule`

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    monitor: platform
    role: alert-rules ❶
  name: sample-rules
  namespace: platform-monitoring
spec:
  groups:
  - name: kube-apiserver.rules
    rules:
```

```
  - expr: | ❷
      sum by (code,resource) (rate(
          apiserver_request_total{job="apiserver",verb=~"LIST|GET"}[5m]
      ))
    labels:
      verb: read
    record: code_resource:apiserver_request_total:rate5m
- name: kubernetes-apps
  rules:
  - alert: KubePodNotReady ❸
    annotations:
      description: Pod {{ $labels.namespace }}/{{ $labels.pod }} has been in a
        non-ready state for longer than 15 minutes.
      summary: Pod has been in a non-ready state for more than 15 minutes.
    expr: |
      sum by (namespace, pod) (
        max by(namespace, pod) (
          kube_pod_status_phase{job="kube-state-metrics",
phase=~"Pending|Unknown"}
        ) * on(namespace, pod) group_left(owner_kind) topk by(namespace, pod) (
          1, max by(namespace, pod, owner_kind)
(kube_pod_owner{owner_kind!="Job"})
        )
      ) > 0
    for: 15m
    labels:
      severity: warning
```

❶ Это метка, на которую ссылается `ruleSelector` в манифесте `Prometheus` из листинга 9.1.

❷ Это пример правила записи для запросов `LIST` и `GET`, поступавших к экземплярам API-сервера Kubernetes на протяжении 5 минут. Оно использует выражение с метрикой `apiserver_request_total`, которая предоставляется API-сервером, и сохраняет новую метрику с именем `code_resource:apiserver_request_total:rate5m`.

❸ Это правило для оповещений. Оно заставит Prometheus послать предупреждение, если какой-либо Pod проведет в состоянии NotReady больше 15 минут.

Как показывает практика, применение Prometheus Operator и этих пользовательских ресурсов для управления серверами Prometheus и их конфигурацией — очень полезный подход, получивший широкое распространение в данной области. Мы настоятельно рекомендуем его тем, кто использует Prometheus в качестве своего основного средства работы с метриками.

Alertmanager

Еще один ключевой компонент — Alertmanager. Это отдельное приложение, которое обрабатывает оповещения и направляет их дежурным инженерам по тому или

иному средству взаимодействия, состоящему из так называемых "получателей". У системы Prometheus есть правила, которые заставляют ее генерировать оповещения в ответ на выполнение каких-то измеряемых условий. Эти оповещения передаются компоненту Alertmanager, который их группирует и удаляет дубликаты, чтобы при возникновении сбоя, затрагивающего сразу несколько реплик или компонентов, люди не получали лавину сообщений. Затем уведомления направляются получателям, указанным в конфигурации. Получатели — это поддерживаемые системы уведомлений, такие как электронная почта, Slack или PagerDuty. Если нужно реализовать систему, которая еще не поддерживается, у Alertmanager на этот случай есть получатель на основе веб-хука, ему можно предоставить URL-адрес, по которому Alertmanager будет посылать POST-запросы с полезными данными в формате JSON.

При использовании Prometheus Operator новый экземпляр Alertmanager можно развернуть с помощью манифеста, как показано в листинге 9.7.

Листинг 9.7. Пример манифеста для ресурса Alertmanager

```
apiVersion: monitoring.coreos.com/v1
kind: Alertmanager
metadata:
  labels:
    alertmanager: main
  name: main
  namespace: platform-monitoring
spec:
  image: quay.io/prometheus/alertmanager:v0.21.0
  nodeSelector:
    kubernetes.io/os: linux
replicas: 2 ❶
securityContext:
  fsGroup: 2000
  runAsNonRoot: true
  runAsUser: 1000
serviceAccountName: alertmanager-main
version: v0.21.0
```

❶ Для развертывания Alertmanager в конфигурации с высокой доступностью можно запросить сразу несколько реплик.

И хотя этот пользовательский ресурс предоставляет очень удобные методы развертывания экземпляров Alertmanager, существует не так уж много кластеров, которым нужно больше одной копии этого компонента, особенно учитывая то, что он может быть развернут в высокодоступной конфигурации. У вас может быть один центральный экземпляр Alertmanager для нескольких кластеров, но выделение одного экземпляра для каждого кластера является разумным решением, которое позволяет сократить для каждого кластера количество внешних зависимостей. Наличие общего экземпляра Alertmanager в рамках отдельно взятого кластера дает

возможность пользователям платформы создавать новые правила для оповещений с помощью единственного ресурса `PrometheusRule`. В такой модели каждый сервер Prometheus настраивается для отправки оповещений компоненту кластера Alertmanager.

Grafana

Чтобы администраторы сложной платформы, основанной на Kubernetes, имели представление о том, что в ней происходит, на основе данных, хранящихся в Prometheus, необходимо генерировать графики и информационные панели. Grafana (**https://grafana.com**) — это слой визуализации с открытым исходным кодом, который стал стандартным решением для просмотра метрик Prometheus. Проект kube-prometheus предоставляет широкий спектр информационных панелей, которые можно взять за основу, не говоря уже о панелях, развиваемых сообществом. И, конечно же, можно создавать собственные графики для вывода хронологических данных из любой системы, которой вы управляете в рамках своей платформы.

Визуализация метрик важна и для разработчиков приложений. Это связано с тем, как вы развертываете свои серверы Prometheus. Если в вашем кластере существует несколько экземпляров этой системы, как вы будете предоставлять собранные метрики пользователям вашей платформы? С одной стороны, добавление информационной панели Grafana в каждый сервер Prometheus может быть удачным решением, обеспечивающим удобное разделение ответственности. С другой стороны, если для просмотра различных информационных панелей придется постоянно заходить на разные серверы, это может быть обременительным для пользователей. В таком случае у вас есть два варианта:

- Использовать федеративный (многокластерный) подход: собрать метрики с разных серверов на едином сервере и затем добавить туда информационную панель, чтобы метрики для разных систем были доступны в одном месте. Подобный метод применяется при работе с такими проектами, как Thanos.
- Добавить несколько источников данных в единую информационную панель Grafana. В этом случае одна информационная панель будет выводить метрики, принадлежащие нескольким серверам Prometheus.

Все сводится к тому, каким образом вы предпочитаете усложнить свою систему: за счет объединения экземпляров Prometheus в федерацию или путем администрирования более сложных конфигураций Grafana. В первом случае необходимо учитывать повышенное потребление ресурсов, но, если вас это устраивает, выбор того или иного метода становится делом вкуса.

Если вы используете в своем кластере единый сервер Prometheus, а администраторы и пользователи вашей платформы обращаются за метриками в одно и то же место, вам следует подумать о правах доступа для просмотра и редактирования информационных панелей. Скорее всего, вам придется сконфигурировать доступ для организаций, групп и пользователей в соответствии со своими требованиями.

Node exporter

Node exporter (**https://github.com/prometheus/node_exporter**) — это агент узла для сбора метрик оборудования и операционной системы, который обычно имеет вид объекта DaemonSet. Он предоставляет статистику уровня хоста для центрального процессора, памяти, дискового ввода/вывода, использования пространства на диске и сетевого трафика, а также информацию о файловых дескрипторах — и это лишь некоторые из метрик, собираемые по умолчанию. Как уже упоминалось, это одно из самых распространенных средств экспорта метрик. Системы Linux не имеют встроенной поддержки формата Prometheus. Node exporter умеет извлекать эти метрики из ядра и предоставлять их в том виде, в котором Prometheus может их собрать. Этот инструмент полезен для мониторинга системы и аппаратного обеспечения с помощью Prometheus на Unix-подобных серверах.

kube-state-metrics

kube-state-metrics (**https://github.com/kubernetes/kube-state-metrics**) предоставляет метрики, относящиеся к целому ряду ресурсов Kubernetes. Это фактически средство экспорта информации о ресурсах, собранной из Kubernetes API. Например, kube-state-metrics дает доступ к таким сведениям, как время запуска, состояние, метки, класс приоритета, запросы ресурсов и лимиты Pod'а. Такую информацию мы обычно получаем с помощью команд `kubectl get` или `kubectl describe`. Эти метрики могут пригодиться для обнаружения серьезных проблем в кластере, например, если Pod вошел в состояние циклических сбоев или пространство имен почти исчерпало свои квоты на ресурсы.

Prometheus adapter

Мы решили включить в этот список проект prometheus adapter (**https://github.com/DirectXMan12/k8s-prometheus-adapter**), так как он входит в состав стека kube-prometheus. Но он не является средством экспорта метрик и не имеет отношения к основной функциональности Prometheus. На самом деле это *клиент* Prometheus. Он извлекает метрики из API-интерфейса этой системы и делает их доступными через API-интерфейсы для работы с метриками в Kubernetes. В результате появляется возможность автоматического масштабирования приложений. Больше об автомасштабировании можно узнать в *главе 13*.

Как видите, существует множество компонентов, позволяющих создавать системы управления метриками и оповещениями. Мы обсудили, как это можно сделать с помощью Prometheus и методик, реализованных в стеке kube-prometheus, включая Prometheus Operator, чтобы облегчить управление этими процессами. Рассмотрение журналирования и метрик закончено. Перейдем к трассировке.

Распределенная трассировка

В общем смысле трассировка означает специализированный способ захвата событий с проходом по пути выполнения. Трассировать можно и отдельный программ-

ный компонент, но в данном разделе мы имеем дело с распределенной трассировкой запросов в микросервисных архитектурах, охватывающей разные приложения. Такая технология приносит огромную пользу организациям, перешедшим на распределенные системы. Мы поговорим о том, как сделать распределенную трассировку доступной на вашей платформе в виде услуги, которой могут пользоваться разработчики приложений.

Важное отличие трассировки от журналирования и метрик: необходимо, чтобы технологии, которые используются приложениями и платформой, были совместимыми. Если журнальные записи выводятся в stdout и stderr, сервисам, отвечающим за их агрегацию, должно быть все равно, как они сохраняются внутри приложения. А такие распространенные метрики, как потребление ресурсов процессора и памяти, можно собирать из приложений без специального инструментария. Но, если приложение задействует клиентскую библиотеку, не совместимую с системой трассировки, которую предлагает платформа, трассировка работать не будет. В связи с этим в данной области большое значение имеет тесное сотрудничество разработчиков платформы и приложений.

Рассматривая тему распределенной трассировки, мы обсудим спецификации OpenTracing и OpenTelemetry, а также некоторые технологии, применяемые в этой области. Мы также пройдемся по компонентам, которые необходимы для трассировки во многих проектах. После этого мы затронем тему инструментирования приложений для поддержки трассировки и последствия использования mesh-сети.

OpenTracing и OpenTelemetry

OpenTracing (**https://opentracing.io**) — это открытая спецификация для распределенной трассировки, которая помогает экосистеме сойтись на общих стандартах реализации. В ее основе лежит три концепции, которые важны для понимания трассировки:

- *Трейс (trace)* — запрос от конечного пользователя распределенной системы, проходит через отдельные сервисы, которые заняты обработкой процесса и участвуют в его выполнении. Трейс представляет всю эту транзакцию и является той сущностью, которую мы хотим проанализировать. Трейс состоит из нескольких спанов.
- *Спан (span)* — каждый отдельный сервис, обрабатывающий запрос. Операции, происходящие в пределах приложения, формируют отдельный спан, который является частью трейса.
- *Тег (tag)* — это метаданные, прикрепленные к спанам для придания им контекста в рамках трейса и предоставления индексов, по которым можно искать.

При визуализации трейсы обычно включают в себя каждый отдельный спан и наглядно показывают, какие компоненты в системе больше всего влияют на производительность. Они также помогают отслеживать участки, в которых возникают ошибки, и понять, как эти ошибки сказываются на работе других компонентов приложения.

Недавно произошло слияние OpenTracing и OpenCensus, результатом которого стал проект OpenTelemetry (**https://opentelemetry.io**). На момент написания этих строк система Jaeger, являющаяся хорошим показателем принятия тех или иных проектов в экосистеме, имеет лишь экспериментальную поддержку OpenTelemetry, но вполне можно ожидать, что в будущем OpenTelemetry станет стандартом де-факто.

Компоненты трассировки

Чтобы предоставить трассировку в качестве одной из услуг платформы, нужно подготовить несколько компонентов. Подход, который мы будем здесь обсуждать, применим к таким проектам с открытым исходным кодом, как Zipkin (**https://zipkin.io**) и Jaeger (**https://www.jaegertracing.io**), но та же модель зачастую подходит и для других проектов и продуктов с коммерческой поддержкой, реализующих стандарты OpenTracing.

Агент

Каждый компонент распределенного приложения будет выводить спан для каждого обрабатываемого запроса. Агент играет роль сервера, которому приложение шлет информацию о спанах. В платформах на основе Kubernetes эту обязанность обычно выполняет агент узла, размещенный на каждом сервере в кластере и принимающий спаны от всех приложений на своем узле. Агент объединяет принятые спаны и передает их центральному сборщику.

Сборщик

Сборщик обрабатывает спаны и сохраняет их в базе данных. Он отвечает за проверку, индексацию и любые преобразования спанов перед их сохранением.

Хранилище

Список поддерживаемых баз данных зависит от проекта, но обычно в него входят Cassandra (**https://cassandra.apache.org**) и Elasticsearch (**https://www.elastic.co/elasticsearch**). Даже в выборочном режиме распределенные системы трассировки собирают огромные объемы информации, и для проведения полезного анализа необходимы базы данных, способные ее обрабатывать и выполнять по ней быстрый поиск.

API-интерфейс

Как можно было бы ожидать, следующий компонент — API-интерфейс, предоставляющий клиентам доступ к сохраненным данным. С его помощью другие приложения и слои визуализации могут обращаться к трейсам и соответствующим спанам.

Пользовательский интерфейс

С этим компонентом пользователи вашей платформы взаимодействуют напрямую. Этот слой визуализации обращается к API-интерфейсу и предоставляет данные

разработчикам приложений. Именно с его помощью инженеры могут просматривать собранную информацию в виде наглядных графиков, анализируя свои системы и распределенные приложения.

Рассмотренные компоненты трассировки, связи между ними и распространенные методы их развертывания проиллюстрированы на рис. 9.8.

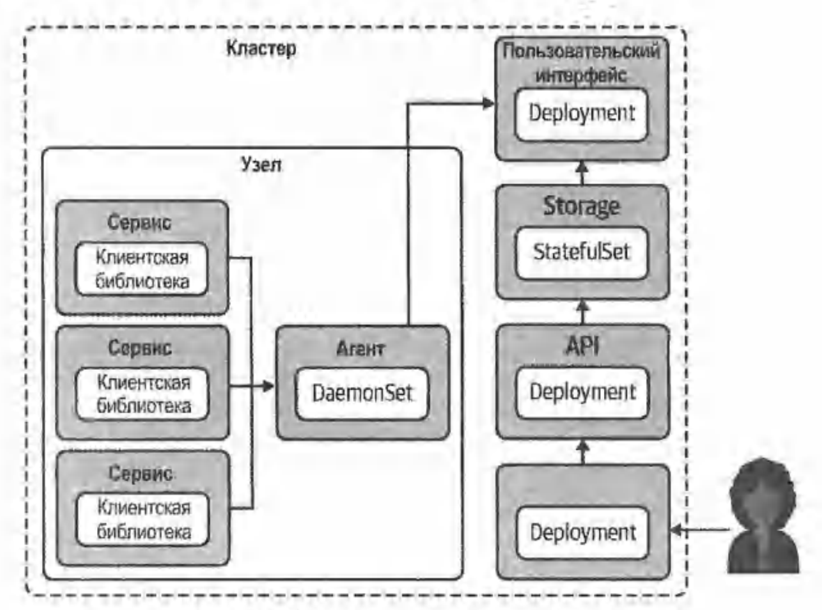


Рис. 9.8. Компоненты сервиса платформы, отвечающего за трассировку

Инструментирование приложений

Чтобы мы могли собирать спаны и объединять их в трейсы, приложение должно доставлять эту информацию. В связи с этим вам необходимо привлечь к данному процессу разработчиков приложения. Даже лучший в мире сервис трассировки будет бесполезен, если приложения не поставляют ему нужные исходные данные. В *главе 14* мы обсудим эту тему более подробно.

Mesh-сети

Если у вас развернута mesh-сеть, то вам, скорее всего, захочется включить данные о ней в результаты своих трассировок. Mesh-сети предоставляют прокси-серверы для запросов, которые принимают и отправляют приложения. Сведения о том, когда и как долго эти серверы обрабатывали тот или иной запрос, помогут вам понять, как они влияют на производительность. Обратите внимание на то, что приложение по-прежнему необходимо инструментировать даже при использовании mesh-сети. Заголовки запросов нужно передавать от одного сервиса к другому при участии трейсов. Mesh-сети были подробно рассмотрены в *главе 6*.

Резюме

Наблюдаемость — один из ключевых аспектов проектирования платформы. Можно утверждать, что ни одну платформу приложений, в которой не решен вопрос наблюдаемости, нельзя считать готовой к примышленному внедрению. У вас всегда должна быть возможность легко собирать журнальные записи из ваших контейнерных приложений и направлять их системе журналирования вместе с журналами аудита, полученными из API-сервера Kubernetes. Среди минимальных требований также желательно иметь поддержку метрик и оповещений. Собирайте метрики, предоставляемые плоскостью управления Kubernetes, выводите их на информационной панели и генерируйте на их основе оповещения. Сотрудничайте с разработчиками, чтобы их приложения предоставляли метрики, если это необходимо, и сделайте так, чтобы они тоже собирались. Наконец, если ваша команда перешла на микросервисную архитектуру, обратитесь к разработчикам приложений и общими усилиями добавьте в их код поддержку трассировки, а также установите компоненты платформы, которые смогут воспользоваться этой информацией. Имея в своем распоряжении такие системы, вы сможете лучше ориентироваться в том, что происходит с вашей платформой. В итоге у вас будет возможность диагностировать и совершенствовать ее работу для улучшения производительности и стабильности.

Идентификация

Идентификация пользователей и приложений — ключевой аспект проектирования и реализации платформы на основе Kubernetes. Вам вряд ли захочется увидеть в новостях сюжет о проникновении злоумышленников в вашу систему. Поэтому крайне важно сделать так, чтобы только сущности с подходящими привилегиями (живые пользователи или приложения) могли обращаться к определенным системам или выполнять определенные действия. Чтобы этого добиться, необходимо реализовать и аутентификацию, и авторизацию:

- *Аутентификация* — это процесс подтверждения подлинности приложения или пользователя.
- *Авторизация* — это процесс определения того, какие действия позволено выполнять приложению или пользователю после того, как они аутентифицировались.

Данная глава целиком посвящена аутентификации. Но это вовсе не означает, что авторизация неважна, и мы будем ее затрагивать там, где это необходимо. Если вы хотите узнать больше, вам непременно следует исследовать систему управления доступом на основе ролей (англ. Role Based Access Control или RBAC) в Kubernetes (на эту тему есть много замечательных ресурсов) и убедиться в том, что у вас есть основательная стратегия ее реализации для собственных приложений. Это поможет вам лучше понять, какие права доступа требуются внешним приложениям, которые вы можете развернуть.

Определение подлинности с целью аутентификации — ключевое требование почти любой распределенной системы. В качестве простого примера можно привести имя пользователя и пароль, с которыми сталкиваемся мы все. Их сочетание идентифицирует вас как пользователя системы. В этой связи у идентифицируемой сущности должно быть два свойства:

- Возможность проверки подлинности. Если пользователь ввел свое имя и пароль, у нас должна быть возможность обратиться к базе данных или другому достоверному источнику информации и убедиться в том, что введены корректные значения. Если нам предоставили сертификат TLS, у нас должна быть возможность проверить его подлинность, обратившись к доверенному центру сертификации (ЦС), который его выдал.
- Уникальность. Если предоставленные нам идентификационные данные не уникальны, то мы не можем однозначно идентифицировать их владельца. Однако уникальность достаточно обеспечивать лишь *в нужных нам рамках*, например, для имени пользователя или адреса электронной почты.

Подтверждение подлинности также является важнейшим условием проведения авторизации. Прежде чем определять, к какому диапазону ресурсов следует выдать доступ, мы должны сначала однозначно идентифицировать сущность, которая проходит аутентификацию в системе.

Кластеры Kubernetes, как правило, мультитенантные, каждый из них обслуживает множество пользователей и групп разработчиков, которые развертывают и администрируют разные приложения. Поддержка мультитенантности в Kubernetes создает определенные трудности (многие из них рассмотрены в этой книге), в число которых входит идентификация. Учитывая сочетание привилегий и ресурсов, которое нам нужно продумать, мы должны позаботиться о многих вариантах развертывания и конфигурации. Разработчикам нужен доступ к своим приложениям. Группы эксплуатации должны иметь доступ ко всем приложениям и, возможно, сервисам платформы. Взаимодействие между приложениями необходимо ограничить. Этот список можно продолжить. Что насчет разделяемых сервисов, отделов безопасности, инструментов для развертывания?

Все это распространенные вопросы, которые существенно усложняют конфигурацию и обслуживание кластера. Не забывайте, эти привилегии нужно еще как-то обновлять. Здесь можно легко ошибиться. Но хорошая новость в том, что у Kubernetes есть механизмы, позволяющие интегрироваться с внешними системами и моделировать идентификацию и управление доступом безопасным образом.

Мы начнем эту главу с того, что обсудим идентификацию *пользователей* и различные методы аутентификации в Kubernetes. Затем речь пойдет о вариантах и способах проверки подлинности *приложений*. Вы увидите, как приложения проходят аутентификацию при обращении к API-серверу Kubernetes (это пригодится для написания инструментов, взаимодействующих с Kubernetes напрямую, таких как операторы). Мы также поговорим о создании уникальных идентификаторов, которые позволят приложениям внутри кластера аутентифицировать друг друга, а также проходить аутентификацию во *внешних* сервисах таких, как AWS.

Идентификация пользователей

Этот раздел посвящен подходам к созданию надежной системы идентификации *пользователей* в пределах одного или нескольких кластеров Kubernetes. В данном контексте мы рассматриваем живого пользователя, который взаимодействует с кластером напрямую (либо через утилиту `kubectl`, либо через API-интерфейс). Свойства, которыми должен обладать процесс идентификации (описанные в предыдущем разделе), распространяются и на пользователей, и на приложения, но некоторые из методов будут отличаться. Например, мы всегда хотим иметь уникальные идентификаторы, подлинность которых можно подтвердить, однако в случае с пользователями это достигается за счет OpenID Connect (OIDC), а приложения могут использовать токены служебной учетной записи (англ. Service Account или SA).

Методы аутентификации

Администраторам Kubernetes доступен целый ряд методов аутентификации, каждый из которых имеет свои сильные и слабые стороны. Придерживаясь главного мотива этой книги, отметим, что вы должны разобраться со своими конкретными сценариями использования, определить, какие из этих методов вам подходят, интегрировать их со своими системами, предоставить удобный пользовательский интерфейс и реализовать меры безопасности, которые требует ваша организация.

В этом разделе мы рассмотрим все методы идентификации *пользователей*, их плюсы и минусы, и опишем некоторые широко распространенные подходы, опыт применения которых у нас имеется. Некоторые из представленных здесь методов относятся к конкретной платформе и привязаны к возможностям, которые предоставляют определенные поставщики облачных решений, а другие подходят для любой платформы. То, насколько хорошо система интегрируется с вашим имеющимся стеком технологий, определенно сыграет роль в решении о том, внедрять ее или нет. Нужно найти баланс между наличием дополнительных возможностей и тем, насколько легко поддерживать интеграцию с существующим стеком.

Помимо идентификации, некоторые методы, описанные здесь, предоставляют шифрование. Например, процесс на основе инфраструктуры открытых ключей (англ. Public Key Infrastructure или PKI) генерирует сертификаты, которые можно использовать для взаимодействия по mTLS. Однако шифрование не является центральной темой данной главы, это всего лишь одно из дополнительных преимуществ некоторых методов идентификации.

Общие секреты

Общий секрет — это уникальный фрагмент (или набор) информации, которым владеет как вызывающая сторона, так и сервер. Например, когда приложению нужно подключиться к базе данных MySQL, оно может осуществить аутентификацию по сочетанию имени пользователя и пароля. Этот метод требует, чтобы обе стороны имели доступ к данному сочетанию в том или ином виде. Вы должны создать в MySQL запись с этой информацией и затем передавать ее всем вызывающим приложениям, которым она может понадобиться. Такой подход показан на рис. 10.1; серверное приложение хранит корректные учетные данные, которые должны быть предоставлены клиентской стороной для получения доступа.

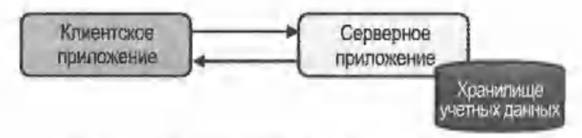


Рис. 10.1. Процесс обмена общими секретами

Kubernetes предлагает два варианта использования модели общих секретов для аутентификации на API-сервере. Первый вариант состоит в том, что вы передаете API-серверу список значений, разделенных запятыми (в формате CVS), которые

привязывают имена пользователей (и, возможно, групп) к статическим токенам. Когда вам нужно аутентифицироваться, вы предоставляете токен носителя (Bearer token) в HTTP-заголовке Authorization. Kubernetes будет считать, что ваш запрос пришел от пользователя, привязанного к токену, и поступит соответствующим образом.

Вы также можете передать API-серверу CVS с именами пользователей (и, возможно, группами) и паролями. В этом случае пользователи должны предоставлять учетные данные в HTTP-заголовке для обычной авторизации (Authorization: Basic), предварительно закодировав их в base64.



В Kubernetes нет ресурса или объекта под названием User или Group. Это просто имена, заранее определенные для идентификации в RBAC RoleBinding. Информация о пользователе может быть взята из статического файла и привязана к токену или паролю (как описывалось ранее), но ее также можно получить из поля `CN` сертификата x509, прочитать из поля `OAuth`, принадлежащего запросу, и т. д. Способ определения пользователя и группы целиком зависит от принятого метода аутентификации, и у Kubernetes нет внутренних механизмов для определения или использования этой информации. Мы считаем, что это является сильной стороной API-интерфейса, так как мы можем подключать разнообразные реализации и делегировать решение этих вопросов системам, которые специально для этого предназначены.

Оба метода обладают серьезными недостатками, и использовать их не рекомендуется. Некоторые из этих недостатков вызваны тем, как они реализованы в Kubernetes, а другие присущи самой модели общих секретов, и мы их вскоре обсудим. Основные недостатки с точки зрения Kubernetes:

- Статические файлы с токенами и/или паролями должны храниться (в открытом виде) в каком-то месте, доступном для API-сервера. Это не так страшно, как может показаться на первый взгляд, ведь если кому-то удастся взломать ваш API-сервер и получить доступ к этому узлу, то незашифрованный файл с паролями будет не самой серьезной из ваших проблем. Тем не менее, процесс установки Kubernetes во многом автоматизирован, и все ресурсы, необходимые для начальной конфигурации, должны храниться в репозитории, который должен быть защищен, проходить аудит и обновляться. В результате появляются дополнительные возможности для утечки учетных данных по небрежности или из-за применения нерекомендуемых подходов.
- Ни у статических токенов, ни у сочетаний имени пользователя и пароля нет сроков годности. Если какие-либо учетные данные скомпрометированы, дыру в безопасности необходимо быстро обнаружить и закрыть, удалив соответствующую учетную информацию и перезапустив API-сервер.
- Внесение любых изменений в эти учетные файлы требует перезапуска API-сервера. На практике (и в отдельно взятом случае) это не составляет труда. Однако многие организации пытаются отказаться от ручного вмешательства в уже запущенное программное обеспечение и активные серверы (и это правильно). В наши дни изменение конфигурации в основном заключается в повторной сборке и развертывании, без входа на серверы по SSH (на этот счет есть извест-

ная аналогия о сельскохозяйственных и домашних животных). Следовательно, изменение конфигурации API-сервера и перезапуск процессов будет, скорее всего, более сложным действием.

Помимо перечисленных недостатков, относящихся непосредственно к Kubernetes, у модели общих секретов есть еще один минус. Если мы имеем непроверенную сущность, как она может *изначально* аутентифицироваться в хранилище секретов, чтобы получить подходящий идентификатор? Эта проблема *безопасного представления* (англ. secure introduction) и ее решение рассматриваются в разделе "Идентификация приложений и рабочих заданий" данной главы.

Инфраструктура открытого ключа



В этом разделе подразумевается, что вы уже знакомы с идеями, лежащими в основе PKI.

Для однозначной идентификации и аутентификации пользователей в Kubernetes в модели PKI предусмотрены сертификаты и ключи. Kubernetes активно применяет PKI для безопасного взаимодействия между всеми основными компонентами системы. Центры сертификации (ЦС) и сертификаты можно настраивать разными способами, но мы продемонстрируем это на примере kubeadm, так как именно этот метод чаще всего встречается в реальных условиях (и фактически является стандартным методом установки для современных версий Kubernetes).

После установки Kubernetes мы обычно получаем файл kubeconfig с информацией о пользователе `kubernetes-admin`. Этот файл фактически является администраторским ключом к кластеру. Обычно он имеет имя admin.conf и выглядит примерно так, как показано в листинге 10.1.

Листинг 10.1

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: <.. ОПУЩЕНО ...>
    server: https://127.0.0.1:32770
  name: kind-kind
contexts:
- context:
    cluster: kind-kind
    user: kind-kind
  name: kind-kind
current-context: kind-kind
kind: Config
preferences: {}
users:
- name: kind-kind
```

```
  user:
    client-certificate-data: <.. ОПУЩЕНО ...>
    client-key-data: <.. ОПУЩЕНО ...>
```

Чтобы определить, от имени какого пользователя мы сможем аутентифицироваться в кластере с помощью этого файла, нужно сначала раскодировать поле `client-certificate-data` (base64) и затем вывести его содержимое с помощью инструмента вроде `openssl` (листинг 10.2).

Листинг 10.2

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 2587742639643938140 (0x23e98238661bcd5c)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=kubernetes
        Validity
            Not Before: Jul 25 19:48:42 2020 GMT
            Not After : Jul 25 19:48:44 2021 GMT
        Subject: O=system:masters, CN=kubernetes-admin
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    <.. ОПУЩЕНО ...>
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Client Authentication
    Signature Algorithm: sha256WithRSAEncryption
        <.. ОПУЩЕНО ...>
```

Мы видим, что этот сертификат был выдан центром сертификации Kubernetes и предназначается для идентификации пользователя `kubernetes-admin` (поле `CN` в `Subject`) из группы `system:masters`. При использовании сертификатов *x*509 любые перечисленные организации (`O=`) воспринимаются кластером как группы, к которым должен принадлежать пользователь. Некоторые нетривиальные методы, связанные с конфигурацией и правами доступа пользователей и групп, будут рассмотрены далее в этой главе.

В представленном примере мы видели стандартную конфигурацию для пользователя `kubernetes-admin`; это имя резервируется по умолчанию и предоставляет администраторские привилегии в пределах всего кластера. Но было бы полезно взглянуть на то, как настраивается выдача сертификатов для идентификации

обычных пользователей системы, которым впоследствии можно назначать подходящие права доступа с помощью RBAC. Выдача и обслуживание сертификатов — трудная задача, но у Kubernetes есть несколько встроенных ресурсов, которые могут ее облегчить.

Для корректной работы процесса CSR, описанного далее, диспетчер контроллеров должен быть запущен с параметрами `--cluster-signing-cert-file` и `--cluster-signing-key-file`:

```
spec:
  containers:
  - command:
    - kube-controller-manager
    - --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt
    - --cluster-signing-key-file=/etc/kubernetes/pki/ca.key
    # Дополнительные флаги опущены для краткости
    image: k8s.gcr.io/kube-controller-manager:v1.17.3
```

Любая сущность с подходящими правами доступа RBAC может обратиться к API-интерфейсу Kubernetes с запросом на получение сертификата (англ. Certificate Signing Request или CSR). Если у пользователя должна быть возможность *самостоятельного обращения с этим запросом*, мы должны предоставить ему соответствующий механизм. Для этого можно явно разрешить отправку и получение CSR для пользователя `system:anonymous` и/или группы `system:unauthenticated`.

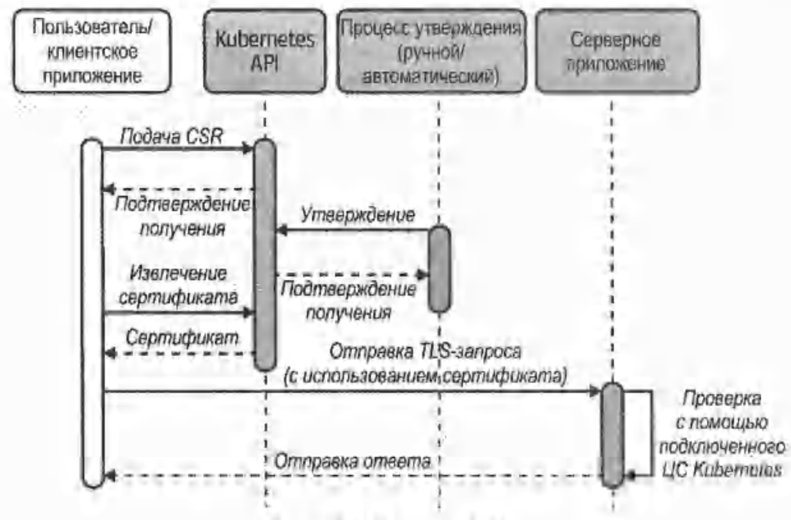


Рис. 10.2. Процесс CSR

Без этого любой пользователь, не прошедший аутентификацию, по определению не сможет инициировать процесс, который бы позволил ему аутентифицироваться. Но к такому подходу определенно стоит относиться с осторожностью, так как неаутентифицированные пользователи не должны иметь никакого доступа к API-серверу Kubernetes. В связи с этим для предоставления возможности самостоятельной отправки CSR обычно используют тонкий слой абстракции или портал поверх

Kubernetes, который выполняется с подходящими правами доступа. Пользователи могут заходить на этот портал с помощью каких-то других учетных данных (обычно SSO) и инициировать процесс CSR (как показано на рис. 10.2).

В рамках этого процесса пользователь может локально сгенерировать закрытый ключ и отправить его через портал. Или портал может генерировать закрытые ключи для каждого пользователя и возвращать их с утвержденными сертификатами. Для генерации можно применять `openssl` или ряд других инструментов/библиотек. Запрос CSR должен содержать метаданные, которые пользователь хочет закодировать в своем сертификате *x*509, включая свое имя и любые дополнительные группы, в которые он должен входить. В следующем примере (листинг 10.3) создается запрос на получение сертификата, который идентифицирует пользователя как *john*.

Листинг 10.3

```
$ openssl req -new -key john.key -out john.csr -subj "/CN=john"
$ openssl req -in john.csr -text
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: CN=john
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (1024 bit)
                Modulus:
                    <.. ОПУЩЕНО ...>
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: sha256WithRSAEncryption
        <.. ОПУЩЕНО ...>
```

После генерации запроса CSR мы можем отправить его кластеру через портал, используя ресурс `CertificateSigningRequest`. В листинге 10.4 показан пример запроса в виде файла YAML, но наш портал, скорее всего, применил бы эту конфигурацию посредством API-интерфейса Kubernetes, не требуя от нас ручного создания документа YAML.

Листинг 10.4

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: john
spec:
  request: $(cat john.csr | base64 | tr -d '\n')
```

```
  usages:
  - client auth
EOF
```

В результате в Kubernetes будет создан объект CSR в состоянии `pending`, ожидающий утверждения. Он будет содержать запрос на получение сертификата (закодированный в base64) и имя пользователя, который его отправил. Если аутентификация в API-интерфейсе Kubernetes осуществляется через токен служебной учетной записи (как это бы сделал Pod в автоматизированном процессе), то в качестве имени пользователя будет выбрано имя этой SA. В следующем примере мы аутентифицировались в API-интерфейсе Kubernetes как пользователь `kubernetes-admin`, и соответствующее имя выводится в поле `Requestor`. Если бы использовался портал, то мы бы увидели назначенную ему служебную учетную запись:

```
$ kubectl get csr
NAME     AGE   REQUESTOR          CONDITION
my-app   17h   kubernetes-admin   Pending
```

Пока запрос ожидает рассмотрения, пользователю не выдается никакой сертификат. На следующем этапе CSR утверждается администратором кластера (или пользователем с подходящими правами доступа). Это тоже можно автоматизировать, если личность пользователя может быть определена программным путем. В случае утверждения запроса пользователю будет возвращен сертификат, с помощью которого можно подтвердить личность в кластере Kubernetes. В связи с этим необходимо убедиться в том, что пользователь, подавший запрос, *действительно* является тем, за кого себя выдает. Для этого в CSR можно включить некоторые дополнительные метаданные и автоматизировать процесс проверки предоставленной информации на принадлежность заявленному пользователю. Но мы также можем воспользоваться для этого внешним механизмом.

После утверждения CSR сертификат можно извлечь (из поля `status` в CSR) и использовать (в сочетании с его закрытым ключом) для взаимодействия с API-интерфейсом Kubernetes по mTLS. В нашей реализации с порталом сертификат загружался бы системой портала и становился бы доступным для запросившего его пользователя сразу после его повторного входа в портал (листинг 10.5).

Листинг 10.5

```
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: my-app
# Остальные поля убраны для краткости
status:
  certificate: <.. ОПУЩЕНО ...>
  conditions:
  - lastUpdateTime: "2020-03-04T15:45:30Z"
    message: This CSR was approved by kubectl certificate approve.
```

```
    reason: KubectlApprove
    type: Approved
```

Декодировав сертификат, мы можем увидеть, что в его поле `CN` содержится подходящая идентификационная информация (о пользователе *john*), как показано в листинге 10.6.

Листинг 10.6

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            66:82:3f:cc:10:3f:aa:b1:df:5b:c5:42:cf:cb:5b:44:e1:45:49:7f
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=kubernetes
        Validity
            Not Before: Mar 4 15:41:00 2020 GMT
            Not After : Mar 4 15:41:00 2021 GMT
        Subject: CN=john
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    <.. ОПУЩЕНО ...>
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Extended Key Usage:
                TLS Web Client Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                EE:8E:E5:CC:98:41:78:4A:AE:32:75:52:1C:DC:DD:D0:9B:95:E0:81
    Signature Algorithm: sha256WithRSAEncryption
        <.. ОПУЩЕНО ...>
```

Наконец, мы можем написать конфигурационный файл для kubeconfig с нашим закрытым ключом и утвержденным сертификатом внутри. Это позволит нам взаимодействовать с API-сервером Kubernetes от имени пользователя *john*. Сертификат, который мы получили на предыдущем этапе CSR, нужно указать в поле `client-certificate-data` файла kubeconfig, как показано в листинге 10.7.

Листинг 10.7

```
apiVersion: v1
clusters:
```

```
- cluster:
    certificate-authority-data: <.. ОПУЩЕНО ...>
    server: https://127.0.0.1:32770
  name: kind-kind
contexts:
- context:
    cluster: kind-kind
    user: kind-kind
  name: kind-kind
current-context: kind-kind
kind: Config
preferences: {}
users:
- name: kind-kind
  user:
    client-certificate-data: <.. ОПУЩЕНО ...>
    client-key-data: <.. ОПУЩЕНО ...>
```

Нам встречались реализации описанной модели в реальных условиях, в которых автоматическая система выдает сертификаты на основе учетных данных SSO, поддающихся проверке, или другого метода аутентификации. Благодаря автоматизации эти системы можно успешно применять, но мы не советуем так делать. Применение сертификатов *x*509 в качестве основного метода аутентификации пользователей в Kubernetes создает ряд проблем:

- Сертификаты, выданные в рамках процесса Kubernetes CSR, нельзя отозвать до истечения их срока действия. В настоящее время Kubernetes не поддерживает списки отзыва сертификатов из протокола OSCP (Online Certificate Status Protocol).
- Помимо создания и обслуживания компонента, отвечающего за выдачу сертификатов на основе внешней аутентификации, необходимо выделять, поддерживать и сопровождать дополнительную инфраструктуру PKI.
- У сертификатов *x*509 есть временные метки, определяющие их срок действия, который должен быть относительно коротким, чтобы снизить риск утечки пары ключ-сертификат. Это приводит к постоянному "круговороту" сертификатов, их нужно регулярно раздавать пользователям, чтобы обеспечить стабильный доступ к кластеру.
- Все, кто запрашивает сертификат, должны проходить какую-то проверку подлинности. В автоматической системе можно предусмотреть механизмы на основе метаданных, проверяемых вне кластера. В отсутствие таких метаданных внешняя проверка подлинности зачастую слишком сильно затягивается, что делает ее непрактичной, особенно учитывая непродолжительный срок действия сертификатов, как уже отмечалось ранее.
- Сертификаты действуют в пределах одного кластера. В реальных же проектах и группах разработки число таких кластеров может исчисляться десятками и сот-

нями. Если каждому кластеру нужно предъявлять уникальные учетные данные, сложность их хранения и обслуживания возрастает в разы. Это плохо влияет на удобство использования системы.



Помните, даже если сертификаты не являются основным методом аутентификации, то вы все равно должны хранить файл admin.conf для kubeconfig в каком-то безопасном месте. Если другие методы аутентификации по какой-то причине станут недоступными, это может послужить запасным решением на случай потери доступа к кластеру.

OpenID Connect (OIDC)

Мы считаем, что лучшим решением для аутентификации и идентификации пользователей в Kubernetes является интеграция с существующей системой или провайдером единого входа. Почти любая организация на сегодня имеет решение вроде Okta, Auth0, Google Account или даже внутреннего сервера LDAP/AD, предоставляющего пользователям единую точку для аутентификации и получения доступа к внутренним системам. Когда речь идет о таком процессе, как аутентификация (в котором важным фактором является безопасность), делегирование сложных аспектов внешним системам будет хорошим выбором, если только у вас нет крайне специфических требований.

У этих систем есть много преимуществ. Они основаны на широко распространенных и поддерживаемых стандартах. Они объединяют в себе все функции управления пользовательскими учетными записями и обеспечивают доступ к единой, хорошо защищенной системе, упрощая обслуживание и удаление учетных записей и прав доступа. Кроме того, при использовании общей платформы OIDC они дают возможность пользователям обращаться к внешним приложениям, не раскрывая им свои учетные данные. Еще одно преимущество состоит в том, что многие кластеры Kubernetes в различных окружениях могут пользоваться единым провайдером идентификации, что уменьшает расхождения в конфигурации кластеров.

Kubernetes поддерживает OIDC напрямую в качестве механизма аутентификации (рис. 10.3). Если ваша компания использует провайдер идентификации, который самостоятельно предоставляет необходимые конечные точки OIDC, то настройка Kubernetes для интеграции с ним не составит труда.

Однако существуют несколько ситуаций, в которых для предоставления расширенных возможностей или повышения удобства использования может понадобиться или пригодиться дополнительный инструментарий. Прежде всего, если у вашей организации есть несколько провайдеров идентификации, необходим агрегатор OIDC. Kubernetes поддерживает определение в своей конфигурации лишь одного провайдера, а агрегатор OIDC может играть роль единого промежуточного звена для нескольких других провайдеров (основанных на OIDC или других методах). Мы имеем богатый опыт успешного применения Dex (проект в рамках CNCF Sandbox; **https://oreil.ly/_maX6**), хотя другие популярные решения вроде Keycloak и UAA предлагают похожие возможности.

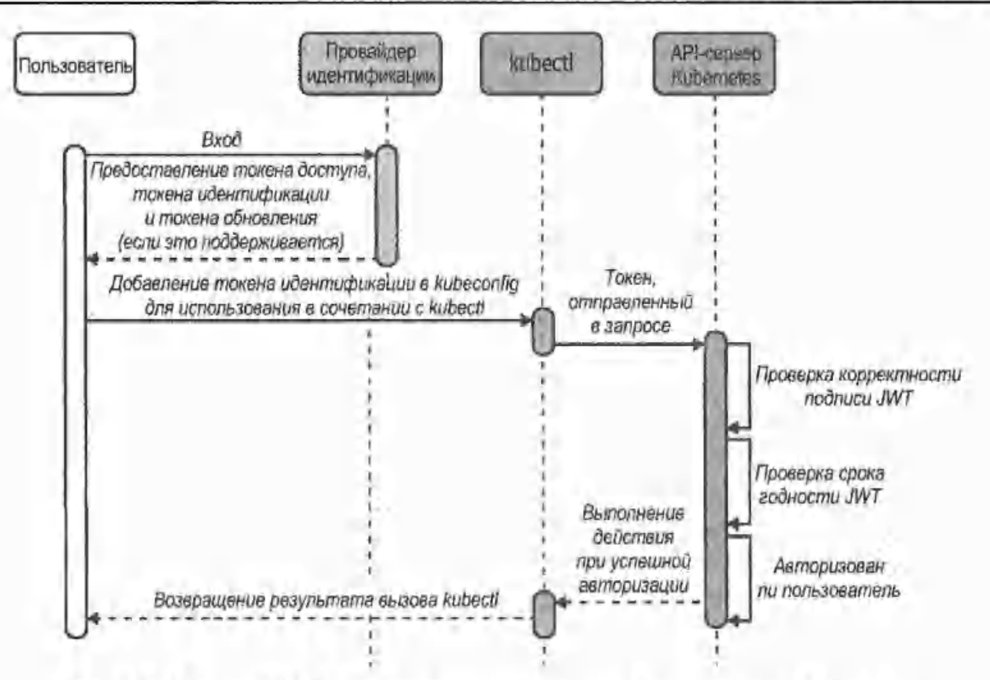


Рис. 10.3. Процесс OIDC. Позаимствован из официальной документации Kubernetes (**https://oreil.ly/VZCz5**)



Не забывайте, что аутентификация является одним из важнейших элементов процесса доступа к кластеру. Dex, Keycloak и UAA можно настраивать в той или иной степени, и при реализации этих решений следует делать упор на доступность и стабильность. Данные инструменты тоже требуют обслуживания, и их необходимо конфигурировать, обновлять и защищать. В своей работе мы всегда пытаемся акцентировать внимание на том, что их добавление делает ваше окружение и кластеры более сложными.

API-сервер можно легко настроить для поддержки OIDC, но вы должны позаботиться о том, чтобы пользователям кластера было удобно работать с этой технологией. Решения на основе OIDC возвращают токен, который нас идентифицирует (в случае успешного входа), но вместе с тем для выполнения операций в кластере нам нужна корректно отформатированная конфигурация kubeconfig. Когда мы в самом начале столкнулись с этим сценарием, наши коллеги разработали веб-интерфейс под названием Gangway, предназначенный для автоматизации процедуры входа в систему через провайдер OIDC и генерации корректного конфигурационного файла kubeconfig на основе возвращенного токена (вместе с необходимыми конечными точками и сертификатами).

Несмотря на то, что OIDC является нашим предпочтительным методом аутентификации, он подходит не для всех случаев, поэтому нам могут понадобиться дополнительные методы. Согласно спецификации, OIDC требует, чтобы пользователь входил в систему непосредственно через веб-интерфейс провайдера идентификации. Это, очевидно, сделано для того, чтобы пользователь гарантированно предоставлял свои учетные данные только доверенному провайдеру, а не приложению. Такое

требование может оказаться проблематичным, если к системе обращается не человек, а программа. Это часто происходит при использовании таких средств автоматизации, как системы CI/CD, которые не умеют передавать учетные данные через веб-интерфейс.

В таких ситуациях нам встречалось несколько разных моделей/решений:

- Если программа привязана к централизованной системе управления учетными записями, мы можем реализовать подключаемый модуль аутентификации для kubectl, который будет входить во внешнюю систему и получать в ответ токен. Kubernetes можно сконфигурировать для проверки этого токена посредством веб-хука. Подобный метод, скорее всего, потребует написания дополнительного кода для создания генератора токенов и сервера, обрабатывающего веб-хук.
- В других ситуациях мы видели, как пользователи возвращались к применению аутентификации на основе сертификатов для программных учетных записей. Этот метод не требует централизованного управления. С другой стороны, вам, естественно, придется управлять процессами выдачи и ротации сертификатов, но зато для этого не нужно создавать собственные компоненты.
- Еще одно ручное, но эффективное решение заключается в создании для программы служебной учетной записи для использования токена, сгенерированного для доступа к API-интерфейсу. Если программа работает *в* кластере, она может задействовать учетные данные, подключенные напрямую к Pod'у. Если программа находится *за пределами* кластера, мы можем вручную скопировать токен в безопасное место, доступное только ей, использовать это для обращения к API-интерфейсу или kubectl. Служебные учетные записи подробно рассматриваются в разделе "Токены служебной учетной записи" данной главы.

Выдача пользователям минимальных привилегий

Итак, мы обсудили, какими способами можно реализовать идентификацию и аутентификацию. Теперь обратимся к смежной теме — авторизации. Мы не станем подробно рассказывать, как вы должны настраивать RBAC в своих кластерах, так как это выходит за рамки данной книги и, вероятно, существенно варьируется в зависимости от конкретных приложений, окружений и групп разработчиков. Тем не менее, мы хотим описать подход на основе принципа минимальных привилегий, который успешно реализуем в реальных условиях.

Какой бы подход вы ни выбрали, однотенантный или мультитенантный, в вашей эксплуатационной команде, скорее всего, будут *суперадминистраторы*, ответственные за конфигурацию, обновление и сопровождение вашей среды. Если у отдельных команд должны быть ограниченные права, обусловленные тем, какой доступ им требуется, эти администраторы будут иметь полный контроль над всем кластером и, следовательно, больше возможностей для выполнения деструктивных действий.

В идеальном мире доступ к кластеру и его эксплуатация были бы автоматизированы с помощью GitOps или, возможно, чего-то подобного. Но на практике мы постоянно

видим, как к кластерам обращаются отдельные пользователи, и, как показывает наш опыт, следующий подход позволяет эффективно ограничить потенциальные проблемы. У вас может возникнуть соблазн назначить роль администратора непосредственно конкретному пользователю по его имени или учетным данным. Но в результате этот пользователь может, к примеру, загрузить не ту конфигурацию kubeconfig и удалить по ошибке что-то важное. Никогда такого не было, и вот опять!

Kubernetes позволяет выполнять операции *от имени другого пользователя*. Благодаря этому мы можем создать механизм, который ведет себя очень похоже на sudo в системах Linux. Он будет ограничивать права, которые пользователь получает по умолчанию, и требовать повышения привилегий для выполнения важных команд. С практической точки зрения мы хотим, чтобы пользователи по умолчанию могли все просматривать, но чтобы для выполнения записи им нужно было сознательно повышать привилегии. Такая модель существенно снижает вероятность возникновения ситуации, описанной ранее.

Давайте посмотрим, как можно было бы реализовать этот метод повышения привилегий. Предположим, что все пользовательские учетные данные, принадлежащие членам эксплуатационной команды, являются частью группы `ops-team` в Kubernetes. Как уже упоминалось ранее, в Kubernetes не существует такого понятия, как группа, поэтому на самом деле имеется в виду, что учетные данные этих пользователей содержат дополнительный атрибут (сертификат *x*509, заявка OIDC и т. д.), который идентифицирует их как участников группы.

Создадим ресурс `ClusterRoleBinding`, с помощью которого пользователи из группы `ops-team` смогут просматривать (`view`) встроенную роль `ClusterRole`, что даст нам наш доступ на чтение по умолчанию (листинг 10.8).

Листинг 10.8

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-view
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: view
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: ops-team
```

Теперь создадим объект `ClusterRoleBinding`, чтобы у нашего пользователя `cluster-admin` были права `ClusterRole`. Помните, мы не привязываем объект `ClusterRole` непосредственно к группе `ops-team`. Ни один пользователь не может быть *напрямую* идентифицирован как `cluster-admin`; это будет пользователь, *от имени которого*

мы выполняем действия, и права которого *принимают* другие аутентифицированные пользователи (листинг 10.9).

Листинг 10.9

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-crb
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: cluster-admin
```

Наконец, создадим роль `ClusterRole` под названием `cluster-admin-impersonator`, которая позволит действовать от имени пользователя `cluster-admin`, и объект `ClusterRoleBinding`, который привяжет ее ко всем членам группы `ops-team` (листинг 10.10).

Листинг 10.10

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-admin-impersonator
rules:
- apiGroups: [""]
  resources: ["users"]
  verbs: ["impersonate"]
  resourceNames: ["cluster-admin"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-impersonate
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin-impersonator
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: ops-team
```

Теперь возьмем конфигурацию kubeconfig для пользователя (*john*) из группы `ops-team`, чтобы увидеть, как повышение привилегий работает на практике:

```
$ kubectl get configmaps
No resources found.
$ kubectl create configmap my-config --from-literal=test=test
Error from server (Forbidden): configmaps is forbidden: User "john"
cannot create resource "configmaps" in API group "" in the namespace "default"
$ kubectl --as=cluster-admin create configmap my-config --from-literal=test=test
configmap/my-config created
```

Мы применили эту конфигурацию для администраторов, хотя реализация чего-то подобного для каждого пользователя является хорошим подходом, который значительно ограничивает возможность допущения серьезных ошибок. Кроме того, одним из преимуществ описанной модели является то, что все это записывается в журнал аудита Kubernetes, поэтому мы можем видеть, как пользователь входит в систему, как он принимает права cluster-admin и затем выполняет действие.

Идентификация контейнерных приложений

В предыдущем разделе мы обсудили основные методы идентификации в Kubernetes "живых" пользователей и показали, как они могут аутентифицироваться в кластере. Здесь же речь пойдет о том, как идентифицировать наши приложения. Мы исследуем три основных сценария:

1. Приложения внутри кластера идентифицируют друг друга, возможно, для выполнения взаимной аутентификации, чтобы усилить безопасность.
2. Приложения идентифицируются для получения нужного доступа к самому API-интерфейсу Kubernetes. Это делают многие пользовательские контроллеры, которым нужно отслеживать ресурсы Kubernetes и что-то с ними делать.
3. Приложения идентифицируются и аутентифицируются для доступа к внешним сервисам. Это может быть что угодно, находящееся за пределами кластера, но на практике это в основном сервисы поставщиков облачных технологий, таких как AWS, GCP и т. д.

В разделе "Сетевая идентификация" данной главы мы рассмотрим два наиболее популярных инструмента для работы с CNI (Container Networking Interface — интерфейса управления сетью контейнеров), Calico и Cilium, и покажем, как они могут назначать учетные данные и ограничивать доступ, в основном в контексте первого сценария использования.

Далее мы перейдем к токенам служебных учетных записей (англ. Service Account Tokens или SAT) и проецируемым токенам служебных учетных записей (англ. Projected Service Account Tokens или PSAT). Это гибкие и важные компоненты Kubernetes, которые позволяют рабочим заданиям идентифицировать друг друга (первый сценарий) и, кроме того, служат основным механизмом идентификации приложений самим API-интерфейсом Kubernetes (второй сценарий).

Затем речь пойдет о ситуациях, в которых идентификацией приложения занимается сама платформа. На практике самым распространенным примером этого являются приложения, которым нужен доступ к сервисам AWS, и мы рассмотрим три главных способа, как это можно реализовать на сегодняшний день.

Напоследок мы поговорим о более общем варианте предыдущего подхода и исследуем инструментарий, предназначенный для предоставления однородной модели идентификации для разных платформ и окружений. Гибкость этого решения позволяет охватить все перечисленные сценарии, и вы увидите, что это может открыть перед нами широкие возможности.

Но, прежде чем приступать к реализации любого из подходов, описанных в этом разделе, вам определенно следует взвесить свои требования в отношении взаимной идентификации приложений. Реализация этой возможности зачастую является задачей повышенной сложности, и большинству организаций это может не понадобиться, по крайней мере, вначале.

Общие секреты

Большинство из того, о чем мы говорили при обсуждении идентификации пользователей с помощью общих секретов, относится и к идентификации приложений, однако есть некоторые дополнительные нюансы и рекомендации, основанные на реальном опыте.

Предположим, мы создали учетные данные, известные клиенту и серверу. Возникает вопрос: как организовать их безопасную ротацию по истечении срока действия? В идеале нам бы хотелось, чтобы эти данные были действительны на протяжении какого-то ограниченного периода. Если они утекут, это уменьшит потенциальный ущерб. Кроме того, поскольку они *общие*, их нужно заново доставлять как клиентскому приложению, так и серверу. Vault от Hashicorp является выдающимся примером хранилища конфиденциальных данных и умеет интегрироваться со множеством инструментов, которые близки к решению этой задачи синхронизации. Однако Vault присуща проблема безопасного представления, с которой мы познакомились в разделе "Идентификация пользователей" данной главы.

Именно эта проблема возникает, когда мы пытаемся обеспечить безопасную доставку учетных данных, как клиенту, так и обслуживающей его сущности еще *до* создания какой-либо модели идентификации и аутентификации (проблема курицы и яйца). Любая первоначальная попытка распределения учетных данных между двумя сущностями может быть скомпрометирована, что нарушает гарантии идентификации и однозначной аутентификации.

Несмотря на уже описанные недостатки, у метода общих секретов есть одно серьезное преимущество, состоящее в том, что его хорошо поддерживают и понимают почти все пользователи и приложения. Это делает его хорошим выбором для обеспечения кроссплатформенной работы. Далее в этой главе вы увидите, как проблему безопасного представления можно решить для Vault и Kubernetes на основе более совершенных методов аутентификации. Если эти методы безопасно сконфигуриро-

вать, Vault становится приемлемым решением (которое мы неоднократно применяли), так как многие из проблем, присущих общим секретам, стоят уже не так остро.

Сетевая идентификация

Сетевые механизмы, такие как IP-адреса, VPN, брандмауэры и т.д., традиционно используются в качестве средств идентификации для определения того, какие приложения имеют доступ к тем или иным сервисам. Однако в облачно-ориентированной экосистеме они себя не оправдывают, и им на смену приходят новые парадигмы. Как показывает наш опыт, эти изменения и то, как к ним можно (и нужно) адаптировать имеющиеся процедуры, необходимо объяснить всем отделам организации (особенно тем, которые отвечают за сеть и безопасность). Очень часто подобная инициатива встречает сопротивление, связанное с вопросами безопасности и/или управления. Но на практике при необходимости можно реализовать почти любую стратегию и вместо того чтобы зацикливаться на аспектах реализации, нужно разобраться в том, какие *на самом деле требования* предъявляет та или иная команда.

В контейнерных окружениях приложения используют общие сетевые стеки и оборудование. Они становятся все более временными и часто перемещаются между узлами. Это приводит к постоянному изменению IP-адресов и сетевой конфигурации.

В многооблачном мире, ориентированном на API-интерфейсы, сеть больше не служит основной границей системы. Мы часто обращаемся к внешним сервисам от разных провайдеров, каждому из которых может понадобиться механизм для проверки подлинности наших вызывающих приложений.

Традиционные сетевые механизмы (IP-адреса хостов, брандмауэры и т.д.), существующие в настоящее время на уровне платформы, не подходят для идентификации приложений, и их можно задействовать только в качестве дополнительного слоя углубленной защиты. Это вовсе не означает, что они *в целом* плохие, просто для их эффективной работы нужен дополнительный контекст, связанный с приложениями. В этом разделе вы увидите, как реализации CNI обеспечивают разные уровни идентификации в кластерах Kubernetes, и как их лучше всего применять. Провайдеры CNI могут контекстуализировать запросы и выполнять идентификацию, сочетая сетевые механизмы с метаданными, полученными из API-интерфейса Kubernetes. Мы проведем краткий обзор некоторых реализаций CNI, пользующихся наибольшей популярностью, и посмотрим, какие возможности они могут предложить.

Calico

Calico (**https://www.projectcalico.org**) обеспечивает соблюдение сетевой политики на сетевом и транспортном уровнях (L3 и L4) модели OSI, благодаря чему пользователи могут ограничивать взаимодействие между Pod'ами с учетом пространств имен, меток и других метаданных. Все эти возможности включаются путем редактирования сетевой конфигурации (`iptables`/`ipvs`) для разрешения/запрета IP-адресов.

Calico также позволяет принимать решения о политике на основе служебных учетных записей с использованием компонента под названием Dikastes в сочетании с прокси-сервером Envoy (развернутым либо отдельно, либо в рамках mesh-сети, как Istio; **https://www.envoyproxy.io** и **https://istio.io**). Этот подход позволяет обеспечивать соблюдение политики на прикладном уровне (L7) с учетом атрибутов протокола приложения (таких как заголовки) и необходимых криптографических учетных данных (например, сертификатов).

Istio (Envoy) по умолчанию работает только по mTLS и следит за тем, чтобы приложения предъявляли сертификаты, подписанные центром сертификации Istio (Citadel). Как видно на рис. 10.3, Dikastes работает в виде sidecar контейнера рядом с прокси-сервером Envoy, развернутым как подключаемый модуль. Envoy проверяет сертификат, прежде чем обращаться к Dikastes, чтобы принять решение о допуске или отклонении запроса. Dikastes принимает это решение на основе таких объектов Calico, как `NetworkPolicy` или `GlobalNetworkPolicy`, определяемых пользователем (листинг 10.11).

Листинг 10.11

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: summary
spec:
  selector: app == 'summary'
  ingress:
    - action: Allow
      source:
        serviceAccounts:
          names: ["customer"]
        NamespaceSelector: app == 'bank'
  egress:
    - action: Allow
```

Согласно данному правилу, данная политика применяется ко всем Pod'ам с меткой `app: summary` и позволяет обращаться к ним только служебной учетной записи `customer` (в пространстве имен с меткой `app: bank`). Это работает, поскольку плоскость управления Calico (агент узла Felix) вычисляет правила путем согласования Pod'ов, запущенных определенной SA с их IP-адресами, с последующей синхронизацией этих сведений с Dikastes через Unix-сокет.

Такая внекластерная сетевая политика играет важную роль, ограничивая потенциальный вектор атаки в окружении Istio. Istio хранит ресурсы PKI каждой служебной учетной записи в объекте Secret, размещенном в кластере. Без этой дополнительной проверки злоумышленник, которому удалось похитить данный объект, смог бы выдать себя за владельца указанной в нем SA (предоставив эти ресурсы PKI), даже если он не использовал ее для входа в систему.

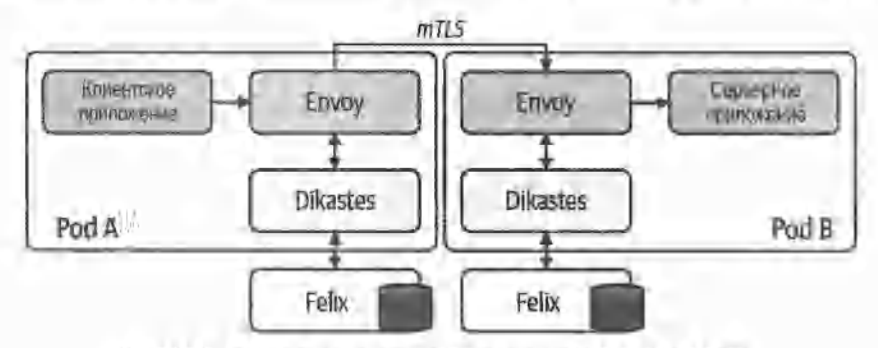


Рис. 10.4. Архитектура, в которой используется Dikastes с Envoy

Если ваша команда уже применяет Calico, то вам определенно следует обратить внимание на компонент Dikastes, так как он может обеспечить дополнительный уровень углубленной защиты. С другой стороны, чтобы сделать его доступным, и чтобы он мог выполняться в вашем окружении, проверяя учетные данные, предоставляемые приложениями, необходимо установить Istio или какое-то другое решение для mesh-сети (например, отдельно стоящий прокси-сервер Envoy). Эти данные не поддаются независимой криптографической проверке, и для каждого подключенного Сервиса необходимо наличие mesh-сети. В результате система серьезно усложняется, поэтому вам следует тщательно взвесить все плюсы и минусы данного подхода. В качестве одного из плюсов можно выделить тот факт, что Calico и Istio являются кроссплатформенными проектами, поэтому данный метод пригоден для идентификации приложений, которые выполняются как в рамках кластера Kubernetes, так и за его пределами (в то время как некоторые решения, рассматриваемые далее, предназначены только для Kubernetes).

Cilium

Cilium (**https://docs.cilium.io**), как и Calico, обеспечивает соблюдение сетевой политики на сетевом и транспортном уровнях, давая возможность пользователям ограничивать взаимодействие между Pod'ами с учетом их пространства имен и других метаданных (например, меток). Cilium также поддерживает (без дополнительных инструментов) возможность применения политики на прикладном уровне и ограничения доступа к Сервисам посредством служебных учетных записей.

В отличие от Calico, работа Cilium не основана на IP-адресах (и обновлении сетевой конфигурации узлов). Вместо этого Cilium идентифицирует каждый уникальный Pod или конечную точку (на основе ряда селекторов) и внедряет соответствующие учетные данные в каждый пакет, предварительно их кодируя. Затем на основе этих данных принимается решение о допуске пакетов на различных этапах их перемещения по системе с использованием хуков ядра eBPF (**https://oreil.ly/Jl9yw**).

Давайте посмотрим, как Cilium проводит идентификацию конечной точки (Pod'а). В листинге 10.12 показан список конечных точек, которые Cilium выводит для приложения. Мы опустили набор меток, но указали дополнительную метку для последнего Pod'а в списке (`deathstar-657477f57d-zzz65`), которой нет у остальных

Pod'ов. В результате мы можем видеть, что последний Pod получает отдельный *идентификатор*. Если не считать этой единственной отличающейся метки, все Pod'ы в объекте `Deployment` имеют общие пространство имен, служебную учетную запись и другие произвольные метки Kubernetes.

Листинг 10.12

```
$ kubectl exec -it -n kube-system cilium-oid9h -- cilium endpoint list
NAMESPACE     NAME                             ENDPOINT ID   IDENTITY ID
default       deathstar-657477f57d-jpzgb       1474          1597
default       deathstar-657477f57d-knxrl       2151          1597
default       deathstar-657477f57d-xw2tr       16            1597
default       deathstar-657477f57d-xz2kk       2237          1597
default       deathstar-657477f57d-zzz65       1             57962
```

Если убрать отличительную метку, то Pod `deathstar-657477f57d-zzz65` будет назначен тот же идентификатор, что и остальным четырем. Такая степень контроля дает нам большие возможности и гибкость при назначении учетных данных отдельным Pod'ам.

Cilium реализует API-интерфейс NetworkPolicy, встроенный в Kubernetes, и по аналогии с Calico предоставляет доступ к более широким возможностям в виде объектов `CiliumNetworkPolicy` и `CiliumClusterwideNetworkPolicy` (листинг 10.13).

Листинг 10.13

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "k8s-svc-account"
spec:
  endpointSelector:
    matchLabels:
      io.cilium.k8s.policy.serviceaccount: leia
  ingress:
  - fromEndpoints:
    - matchLabels:
        io.cilium.k8s.policy.serviceaccount: luke
    toPorts:
    - ports:
      - port: '80'
        protocol: TCP
      rules:
        http:
        - method: GET
          path: "/public$"
```

В рассмотренном примере у нас имеются специальные метки-селекторы `io.cilium.k8s.policy.*`, предназначенные для определенных служебных учетных записей в кластере. Cilium использует свой реестр учетных данных (который мы видели ранее) для открытия/закрытия доступа. Согласно этой политике, мы ограничиваем доступ к пути `/public` на порту 80 для Pod'ов с SA `leia`. Доступ открыт только Pod'ам со служебной учетной записью `luke`.

Cilium, как и Calico, является кроссплатформенным инструментом, поэтому его можно применять в окружениях с Kubernetes и без этой платформы. Для проверки подлинности учетных данных Cilium должен присутствовать в каждом подключенном Сервисе. В результате этот подход может в целом усложнить вашу сетевую конфигурацию. С другой стороны, для работы Cilium не требуется наличие mesh-сети.

Токены служебной учетной записи



Служебные учетные записи — это механизмы Kubernetes, обеспечивающие идентификацию групп Pod'ов. Каждый Pod выполняется от имени SA. Если администратор не создаст такую запись заранее и не назначит ее Pod'у, то будет принята служебная учетная запись, которая по умолчанию задана в том же пространстве имен.

В Kubernetes токены SA имеют формат JWT (JSON Web Token) и создаются в виде объектов `Secret`. Каждая служебная учетная запись (включая ту, что назначена по умолчанию) имеет соответствующий объект `Secret`, содержащий JWT. В стандартной конфигурации эти токены подключаются к каждому Pod'у, запущенному от имени этой SA, и могут использоваться для выполнения запросов к API-интерфейсу Kubernetes (и других сервисах, как будет показано в этом разделе).

Служебные учетные записи Kubernetes позволяют назначить учетные данные группе рабочих заданий. Затем в рамках кластера можно применить правила RBAC (Role-Based Access Control — управление доступом на основе ролей), чтобы ограничить область доступа для отдельно взятой SA. Этот механизм обычно используется самой платформой Kubernetes для аутентификации внутрикластерного доступа к API-интерфейсу:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
  namespace: default
secrets:
- name: default-token-mf9v2
```

Вместе со служебной учетной записью создается связанный с ней объект `Secret`, содержащий уникальный токен JWT, который ее идентифицирует (листинг 10.14).

Листинг 10.14

```
apiVersion: v1
data:
  ca.crt: <.. ОПУЩЕНО ...>
```

```
  namespace: ZGVmYXVsdA==
  token: <.. ОПУЩЕНО ...>
kind: Secret
metadata:
  annotations:
    kubernetes.io/service-account.name: default
    kubernetes.io/service-account.uid: 59aee446-b36e-420f-99eb-a68895084c98
  name: default-token-mf9v2
  namespace: default
type: kubernetes.io/service-account-token
```

По умолчанию к Pod'ам подключается токен служебной учетной записи `default`, действующий в их пространстве имен. Это поведение можно (и нужно) отключить (**https://oreil.ly/kX5mI**), чтобы все SA подключались к Pod'ам явным образом, и чтобы они имели четко определенную и понятную область доступа (вместо того чтобы полагаться на конфигурацию по умолчанию).

Чтобы указать служебную учетную запись для Pod'а, используйте в спецификации последней поле `serviceAccountName`:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: my-pod-sa
# Остальные поля убраны для краткости
```

В результате объект `Secret` служебной учетной записи (с токеном внутри) будет подключен к директории /var/run/secrets/kubernetes.io/serviceaccount/ Pod'a. Приложение может извлечь токен и использовать его для обращения к другим приложениям/сервисам в кластере.

Приложение, к которому обращаются, может проверить предоставленный токен с помощью API-интерфейса Kubernetes `TokenReview` (листинг 10.15).

Листинг 10.15

```
curl -X "POST" "https://<kubernetes API IP>:<kubernetes API Port>\
/apis/authentication.k8s.io/v1/tokenreviews" \
     -H 'Authorization: Bearer <token>' \ ❶
     -H 'Content-Type: application/json; charset=utf-8' \
     -d $'{
  "kind": "TokenReview",
  "apiVersion": "authentication.k8s.io/v1",
  "spec": {
    "token": "<token to verify>" ❷
  }
}'
```

❶ Этот токен представляет собой объект `Secret`, подключенный к Pod'у приложения, которое принимает запрос, что позволяет ей взаимодействовать с API-сервером.

❷ Это тот токен, который вызывающее приложение предъявило в качестве доказательства подлинности.

API-интерфейс Kubernetes вернет в ответ метаданные о токене, который нужно проверить, сообщит, прошел ли тот аутентификацию (листинг 10.16).

Листинг 10.16

```
{
  "kind": "TokenReview",
    "apiVersion": "authentication.k8s.io/v1",
    "metadata": {
        "creationTimestamp": null
    },
    "spec": {
        "token": "<token to verify>"
    },
    "status": {
        "authenticated": true,
        "user": {
            "username": "system:serviceaccount:default:default",
            "uid": "4afdf4d0-46d2-11e9-8716-005056bf4b40",
            "groups": [
                "system:serviceaccounts",
                "system:serviceaccounts:default",
                "system:authenticated"
            ]
        }
    }
}
```

Описанный процесс показан на рис. 10.5.

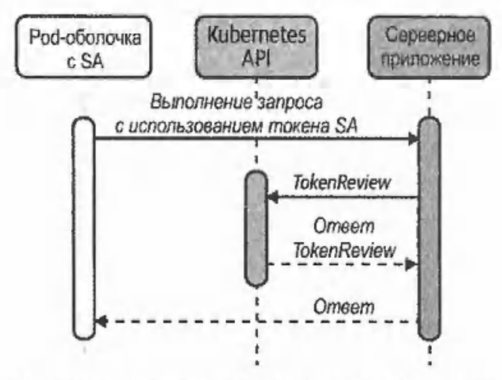


Рис. 10.5. Токены служебной учетной записи

Токены служебной учетной записи входят в состав платформы Kubernetes с самых ранних ее версий и позволяют с ней тесно интегрироваться с помощью удобного формата (JWT). Как администраторы платформы, мы имеем довольно жесткий контроль за их корректностью, так как при удалении SA или объекта `Secret` они становятся недействительными. Однако они обладают определенными свойствами, которые делают их не лучшим выбором в качестве идентификаторов. Прежде всего, токены привязаны к определенной служебной учетной записи и, следовательно, не могут подтверждать подлинность более низкоуровневых компонентов, например, Pod'ов или отдельных контейнеров. К тому же, если мы хотим использовать и проверять токены для идентификации клиентов, потребуется расширить возможности наших приложений. Для этого необходимо создать отдельный компонент, который будет обращаться к API-интерфейсу TokenReview.

Кроме того, токены SA действуют в рамках отдельно взятого кластера, поэтому, если токен был выдан одним кластером, мы не можем использовать его в качестве удостоверяющего документа для внешних сервисов, не открывая доступ к API-интерфейсу TokenReview каждого кластера и не внедряя какие-то дополнительные метаданные о кластере, из которого пришел запрос. Все это существенно усложняет систему, поэтому мы бы советовали отказаться от такого подхода при межкластерной идентификации/аутентификации сервисов.



Чтобы обеспечить выдачу приложениям прав доступа на подходящем уровне, необходимо создавать уникальные служебные учетные записи для каждого приложения, которому нужен доступ к API-серверу Kubernetes. Если же приложению такой доступ *не требуется*, отмените подключение токена SA, указав поле `automountServiceAccountToken: false` в объекте `ServiceAccount`.

Например, это можно сделать для SA по умолчанию, чтобы предотвратить автоматическое подключение токена с учетными данными в соответствующем пространстве имен. Данное поле также можно указывать в объекте `Pod`, но имейте в виду, что в таком случае у него будет более высокий приоритет, если оно указано в обоих местах.

Прогнозируемые токены служебной учетной записи

Начиная с версии v1.12, Kubernetes поддерживает дополнительный метод идентификации, основанный на концепциях токенов служебной учетной записи, но предназначенный для устранения некоторых его слабых мест (таких, как отсутствие TTL, широкая область действия и постоянство).

Для корректной работы прогнозируемых токенов служебной учетной записи (англ. Projected Service Account Tokens или PSAT) нужно сконфигурировать ключи, которые передаются в виде параметров API-серверу Kubernetes, как показано в листинге 10.17 (все они могут быть изменены).

Листинг 10.17

```
spec:
  containers:
  - command:
    - kube-apiserver
    - --service-account-signing-key-file=/etc/kubernetes/pki/sa.key
    - --service-account-key-file=/etc/kubernetes/pki/sa.pub
    - --service-account-issuer=api
    - --service-account-api-audiences=api
    # Остальные флаги убраны для краткости
    image: k8s.gcr.io/kube-apiserver:v1.17.3
```

Процесс идентификации похож на тот, который применяется в методе SAT. Но в данном случае наши Pod'ы/приложения не читают токен SA, подключенный автоматически, вместо этого мы должны подключить прогнозируемый токен SA в виде тома. В результате токен внедряется в Pod, но при этом вы можете указать TTL и то, для кого этот токен предназначен (листинг 10.18).

Листинг 10.18

```
apiVersion: v1
kind: Pod
metadata:
  name: test
  labels:
    app: test
spec:
  serviceAccountName: test
  containers:
  - name: test
    image: ubuntu:bionic
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
    volumeMounts:
    - mountPath: /var/run/secrets/tokens
      name: app-token
  volumes:
  - name: app-token
    projected:
      sources:
      - serviceAccountToken:
          audience: api ❶
          expirationSeconds: 600
          path: app-token
```

❶ Поле `audience` играет важную роль, не давая тем, к кому обращается вызывающее приложение, выдавать себя за него и пользоваться токеном. Для токенов всегда

нужно выбирать подходящую область действия с учетом того, кому направлены вызовы. В данном случае мы ограничиваемся взаимодействием с самим API-сервером.



При работе с PSAT нужно создать и задействовать отдельную служебную учетную запись. Kubernetes не подключает PSAT для SA, которые по умолчанию заданы в пространствах имен.

Вызывающее приложение может прочитать прогнозируемый токен и с его учетом выполнить запросы внутри кластера. Принимающие приложения могут проверить подлинность токена, отправив его API-интерфейсу `TokenReview`. В случае с PSAT `TokenReview` также позволяет убедиться в том, что срок годности токена (TTL) еще не истек, и возвращает дополнительные метаданные о вызывающем приложении, включая информацию о конкретном Pod'е. Эта область действия более узкая, чем в методе SAT, который проверяет только служебную учетную запись:

```
// Остальные поля убраны для краткости
  "extra": {
  "authentication.kubernetes.io/pod-name": ["test"],
  "authentication.kubernetes.io/pod-uid":
    ["8b9bc1be-c71f-4551-aeb9-2759887cbde0"]
}
```

Как видно из рис. 10.6, сами процессы идентификации SAT и PSAT фактически ничем не отличаются (если не считать того, что сервер проверяет поле `audience`); расхождения есть только в проверке действительности токена и его области действия. Поле `audience` имеет большое значение, так как с его помощью можно идентифицировать предполагаемого получателя токена. Если придерживаться официальной спецификации JWT (**https://oreil.ly/gKlA7**), API-интерфейс отклонит токен, получатель которого не соответствует тому, что указано в поле `audience` в конфигурации API-сервера.

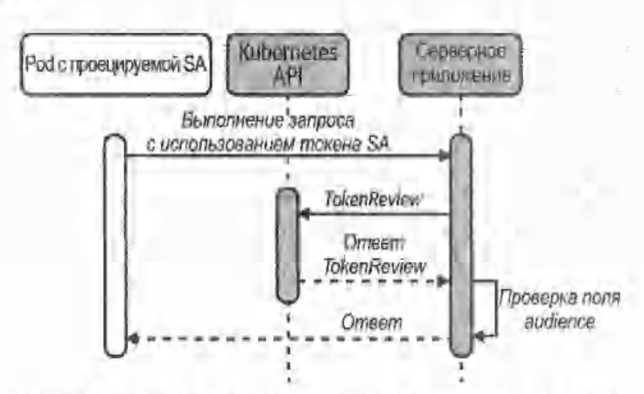


Рис. 10.6. Прогнозируемые токены служебной учетной записи

Прогнозируемые токены служебной учетной записи появились относительно недавно, но оказались чрезвычайно мощным пополнением в арсенале Kubernetes. Они сами по себе обеспечивают тесную интеграцию с платформой, предоставляют настраиваемые TTL и имеют узкую область действия (отдельные Pod'ы). Их также

можно применять в качестве составных элементов для создания более надежных методов (как вы увидите в последующих разделах).

Идентификация узлов на уровне платформы

В ситуациях, когда все приложения выполняются на единой платформе (например, AWS), можно сделать так, чтобы эта платформа сама занималась определением и назначением учетных данных, поскольку она уже обладает контекстными метаданными о приложениях.

Данные для идентификации не предоставляются самим приложением, а определяются на основе его свойств внешним провайдером. Провайдер возвращает приложению учетные данные для подтверждения подлинности, с помощью которых оно может взаимодействовать с другими сервисами платформы. После этого другие сервисы могут легко проверять эти учетные данные, так как они работают поверх той же платформы.

Если взять AWS, инстанс EC2 может запросить учетные данные для подключения к другому сервису, такому как корзина S3. Платформа AWS анализирует метаданные инстанса и может предоставить ему учетные данные с определенной ролью, с помощью которых тот сможет установить соединение. Это показано на рис. 10.7.



Рис. 10.7. Идентификация на уровне платформы



Помните, что платформа по-прежнему должна выполнить *авторизацию* запроса и убедиться в том, что используемые учетные данные имеют подходящие права доступа. Этот метод служит лишь для *аутентификации* запросов.

Возможности, описанные в данном разделе, предоставляются многими поставщиками облачных технологий. Мы решили сосредоточиться на инструментарии, который имеет отношение и интегрируется с Amazon Web Services (AWS), так как в реальных проектах этот поставщик встречается нам чаще всего.

Методы и средства аутентификации на платформе AWS

AWS предоставляет мощное средство идентификации на уровне узла с использованием API-интерфейса метаданных EC2. Это пример системы, в которой платформа (AWS) умеет идентифицировать вызывающую сущность на основе ряда ее характеристик и при этом сама сущность не должна делать никаких утверждений о своей учетной записи. Платформа может передать инстансу безопасные учетные данные (в виде роли, например), которые позволят ей обращаться к сервисам, определен-

ным в рамках соответствующих политик. Весь этот процесс целиком называется управлением учетными данными и доступом (англ. Identity and Access Management или IAM).

Описанная модель лежит в основе того, как AWS (и многие другие поставщики) предоставляют безопасный доступ к своим облачным услугам. Однако с популяризацией контейнеров и других мультитенантных методов развертывания эта система идентификации/аутентификации уровня узла работает плохо, требуя использования дополнительных инструментов и альтернативных подходов.

Далее будут рассмотрены три основных инструмента, которые нам встречаются в реальных условиях. Мы обсудим kube2iam и kiam, два разных проекта с примерно одинаковой моделью реализации (и следовательно, с похожими преимуществами и недостатками). Мы также объясним, почему мы их на данный момент не рекомендуем, и почему вам следует обратить внимание на более интегрированные решения, один из примеров которых мы обсудим последним — роли IAM для служебных учетных записей (англ. IAM Roles for Service Accounts или IRSA).

kube2iam

Проект kube2iam (**https://github.com/jtblin/kube2iam**) — это инструмент с открытым исходным кодом, который выступает прокси-сервером между активными приложениями и API-интерфейсом метаданных AWS EC2. Его архитектура показана на рис. 10.8.



kube2iam требует, чтобы каждый узел в кластере был способен принимать все роли, которые могут понадобиться Pod'ам. Применение этой модели безопасности означает, что в случае взлома контейнера к его содержимому можно получить неограниченный доступ. В связи с этим настоятельно рекомендуется избегать использования kube2iam. Мы обсуждаем этот инструмент, потому что он регулярно встречается в нашей работе, и, прежде чем подробно рассматривать эту реализацию, мы хотим, чтобы вы знали о ее ограничениях.

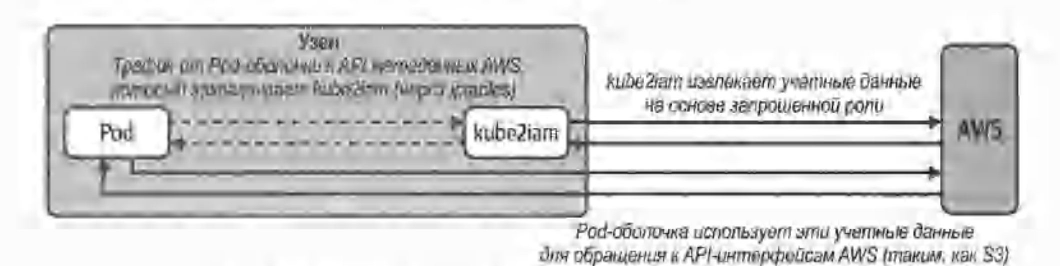


Рис. 10.8. Архитектура и перемещения данных в kube2iam

Pod'ы kube2iam развертываются на каждом узле посредством DaemonSet. Каждая из них внедряет правило iptables для захвата исходящего трафика, направленного к API-интерфейсу метаданных, и перенаправляет его к активному экземпляру kube2iam на том же узле.

Если Pod хочет взаимодействовать с API-интерфейсами AWS, он должен иметь в своей спецификации аннотацию, определяющую роль, которую он хочет получить.

Например, в следующей спецификации Deployment (листинг 10.19) можно видеть роль, указанную в аннотации `iam.amazonaws.com/role`.

Листинг 10.19

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      annotations:
        iam.amazonaws.com/role: <role-arn>
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.9.1
        ports:
        - containerPort: 80
```

Kiam

Kiam (**https://github.com/uswitch/kiam**), как и kube2iam, — это инструмент с открытым исходным кодом, который выступает прокси-сервером для API-интерфейса метаданных AWS EC2, хотя, как видно на рис. 10.9, его архитектура (и, как следствие, модель безопасности) немного отличается в лучшую сторону.



Несмотря на лучшую защищенность по сравнению с kube2iam, kiam имеет одну потенциально серьезную проблему. В этом разделе описывается, как ее минимизировать, но вы все равно должны относиться с осторожностью к использованию kiam и понимать возможные векторы атаки.

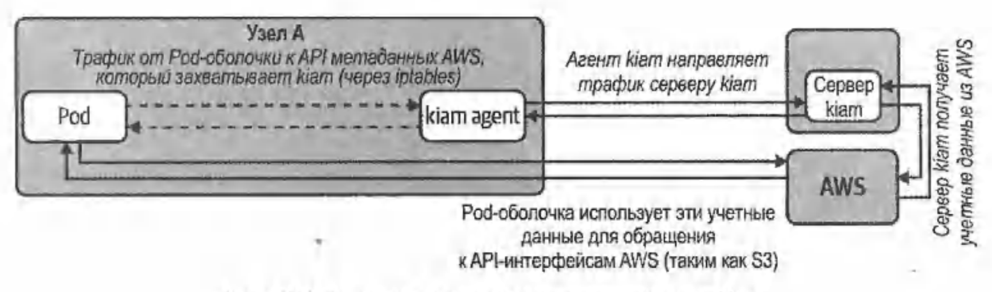


Рис. 10.9. Архитектура и перемещения данных в kiam

Инструмент kiam состоит из двух компонентов: сервера и агента. Агенты выполняются на каждом узле кластера в виде DaemonSet. Серверный компонент может (и должен) быть ограничен узлами управляющей плоскости или каким-то подмножеством узлов кластера. Агенты перехватывают запросы к API-интерфейсу метаданных EC2 и направляют их серверному компоненту, чтобы тот завершил соответствующий процесс аутентификации в AWS. Как видно из рис. 10.10, только серверным узлам нужны права для принятия ролей IAM (опять же, совокупности всех ролей, которые могут потребоваться Pod'ам).

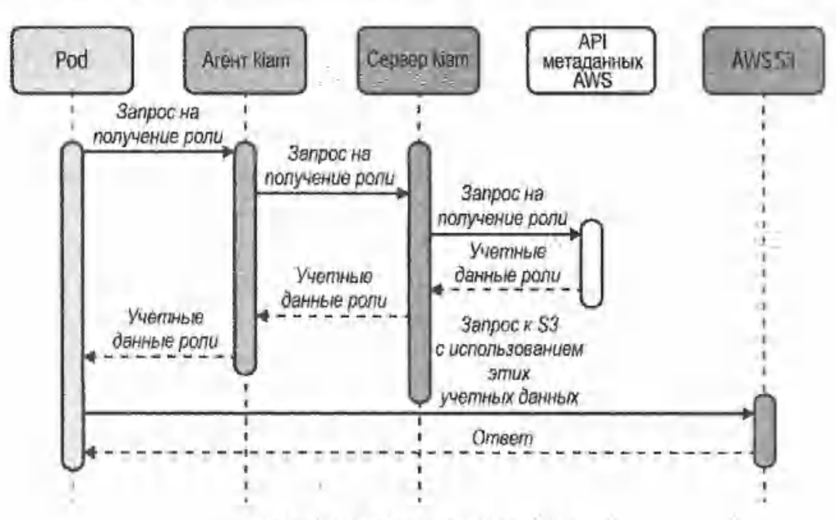


Рис. 10.10. Принцип работы kiam

В этой модели должны быть предусмотрены механизмы, следящие за тем, чтобы приложения могли выполняться на серверных узлах (и, следовательно, получать свободный доступ к API-интерфейсам AWS). Как и в kube2iam, роль, которая должна быть принята Pod, должна быть указана в ее аннотации (листинг 10.20).

Листинг 10.20

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      annotations:
        iam.amazonaws.com/role: <role-arn>
      labels:
        app: nginx
```

```
spec:
  containers:
  - name: nginx
    image: nginx:1.9.1
  ports:
  - containerPort: 80
```

Несмотря на лучшую модель безопасности по сравнению с kube2iam, у kiam тоже есть потенциальная уязвимость: если пользователь может напрямую развернуть Pod на узле (указав в ее спецификации поле `nodeName` в обход планировщика Kubernetes и любых потенциальных средств защиты), он получит неограниченный доступ к API-интерфейсу метаданных EC2.

Смягчить эту проблему можно с помощью изменяющего и проверяющего веб-хуков допуска, которые будут следить за тем, чтобы поле `nodeName` не было указано заранее в спецификации Pod'а, и обновлять запросы к API-интерфейсу Kubernetes.

kiam предоставляет надежный механизм для обеспечения отдельным Pod'ам доступа к API-интерфейсам AWS с помощью модели принятия ролей, уже знакомой существующим пользователям этого облака. Такое решение подходит для множества ситуаций при условии наличия мер предосторожности, описанных ранее.

Роли IAM для служебных учетных записей

С конца 2019 года в AWS существует встроенная интеграция между Kubernetes и механизмом IAM под названием IRSA (IAM Roles for Service Accounts — роли IAM для служебных учетных записей; **https:// oreil.ly/dUoJJ**).

В целом по своим возможностям IRSA напоминает kiam и kube2iam, в том смысле, что пользователь может назначать своим Pod'ам роли AWS IAM с помощью аннотаций. Но реализуется это совсем иначе, благодаря чему исключаются угрозы безопасности, свойственные предыдущим двум инструментам.

AWS IAM поддерживает федеративную идентификацию с использованием стороннего провайдера OIDC, в данном случае API-сервера Kubernetes. Как вы уже видели на примере PSAT, платформа Kubernetes способна создавать и подписывать токены с коротким сроком действия для отдельных Pod'ов.

AWS IRSA сочетает эти возможности с дополнительным провайдером учетных данных, который доступен в официальных SDK и вызывает `sts:AssumeRoleWithWebIdentity`, передавая PSAT. PSAT и нужную роль необходимо внедрить в Pod в виде переменной окружения (существует веб-хук, который делает это автоматически, в зависимости от значения в поле `serviceAccountName`), что иллюстрирует листинг 10.21.

Листинг 10.21

```
apiVersion: apps/v1
kind: Pod
metadata:
```

```
  name: myapp
spec:
  serviceAccountName: my-serviceaccount
  containers:
  - name: myapp
    image: myapp:1.2
    env:
    - name: AWS_ROLE_ARN
      value: "arn:aws:iam::123456789012:role/\
        eksctl-irptest-addon-iamsa-default-my-\
        serviceaccount-Role1-UCGG6NDYZ3UE"
    - name: AWS_WEB_IDENTITY_TOKEN_FILE
      value: /var/run/secrets/eks.amazonaws.com/serviceaccount/token
    volumeMounts:
    - mountPath: /var/run/secrets/eks.amazonaws.com/serviceaccount
        name: aws-iam-token
        readOnly: true
  volumes:
  - name: aws-iam-token
    projected:
      defaultMode: 420
      sources:
      - serviceAccountToken:
          audience: sts.amazonaws.com
          expirationSeconds: 86400
          path: token
```

У Kubernetes нет стандартного механизма, предоставляющего общеизвестную конечную точку OIDC, поэтому для настройки данной технологии в общедоступном месте (таком, как статическая корзина S3) нужно приложить дополнительные усилия и позаботиться о том, чтобы сервис AWS IAM мог проверять токены с помощью открытого ключа служебной учетной записи Kubernetes.

По окончании проверки AWS IAM отвечает на запрос приложения, возвращая вместо PSAT учетные данные запрошенной роли IAM. Это показано на рис. 10.11.

Несмотря на слегка громоздкую настройку, IRSA предлагает лучшую модель безопасности среди всех методов назначения ролей IAM Pod'ам.

Технология IRSA — хороший выбор для организаций, которые уже пользуются услугами AWS, так как в ней применяются методики и базовые элементы, знакомые вашим группам эксплуатации и разработки. Она основана на простом и понятном подходе (привязке служебных учетных записей к ролям IAM), который имеет надежную модель безопасности.

Главный недостаток заключается в том, что конфигурация и развертывание IRSA могут быть несколько громоздкими, если не использовать Amazon EKS (Elastic Kubernetes Service), хотя последние нововведения в самой платформе Kubernetes

смягчают некоторые технические трудности, такие как предоставление Kubernetes в качестве провайдера OIDC.

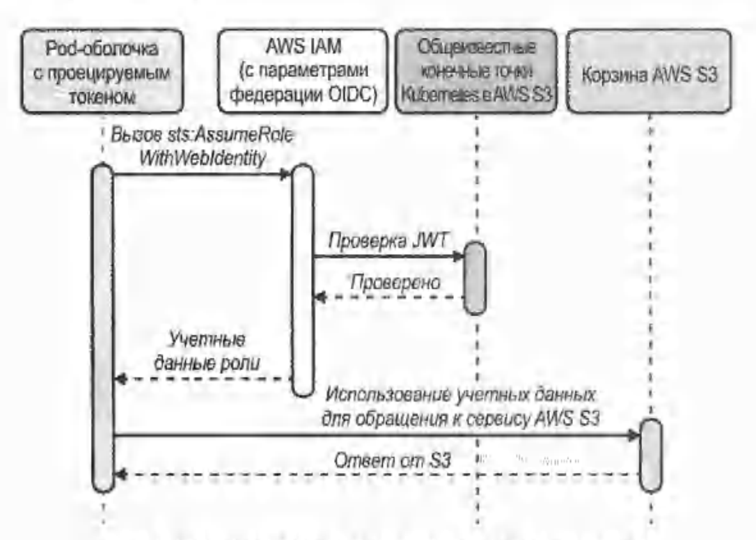


Рис. 10.11. Роли IAM для служебных учетных записей

Как мы уже видели в этом разделе, выполнение идентификации с использованием общей платформы (в данном случае AWS) имеет много преимуществ. В следующем разделе речь пойдет об инструментарии, предназначенном для реализации этой же модели, но способном работать на *разных* платформах. Таким образом мы получаем уровень контроля, присущий централизованной системе идентификации, и гибкость, позволяющую применять ее к любым приложениям в любом облаке и на любой платформе.

Кроссплатформенная идентификация с использованием SPIFFE и SPIRE

SPIFFE (Secure Production Identity Framework for Everyone — безопасная платформа идентификации для всех) — это стандарт, описывающий синтаксис удостоверения (SPIFFE Verifiable Identity Document, SVID) с возможностью задействовать существующие криптографические форматы, такие как *x*509 и JWT. В нем также описывается ряд API-интерфейсов для предоставления и потребления этих удостоверений. Идентификатор SPIFFE имеет вид `spiffe://trust-domain/hierarchical/workload`, где все элементы, идущие за `spiffe://`, являются произвольными строковыми идентификаторами, которые можно использовать различными способами (хотя чаще всего с их помощью создают некую иерархию).

SPIRE (SPIFFE Runtime Environment — среда выполнения SPIFFE) — это эталонная реализация SPIFFE, имеющая ряд пакетов разработки и средств интеграции, с помощью которых приложения могут пользоваться удостоверениями SVID (как предъявлять, так и потреблять).

В этом разделе подразумевается, что SPIFFE применяется в сочетании со SPIRE (если не указано обратное).

Архитектура и концепции

У SPIRE есть серверный компонент, выступающий заверителем подписи для удостоверений и хранящий реестр удостоверений всех рабочих заданий, а также условия, выполнение которых необходимо для их выдачи.

Агенты SPIRE выполняются на каждом узле в виде DaemonSet и предоставляют API-интерфейс, с помощью которого рабочие задания могут запрашивать удостоверения через Unix-сокет. Агент также имеет возможность обращаться к kubelet за метаданными Pod'ов на своем узле. Архитектура SPIRE показана на рис. 10.12.

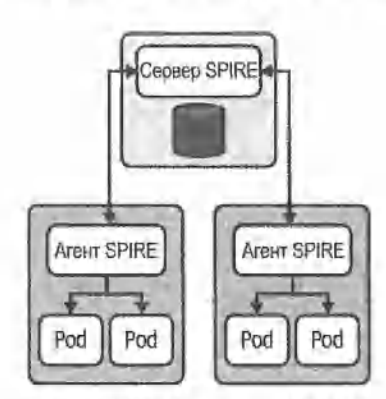


Рис. 10.12. Архитектура SPIRE.
Рисунок позаимствован из официальной документации (**https://oreil.ly/6VY4A**)

После запуска агенты проходят проверку на сервере и регистрируются с помощью процесса, который называется *аттестация узлов* (как показано на рис. 10.13). Этот процесс идентифицирует узел и назначает ему идентификатор SPIFFE, используя контекст окружения (например, API-интерфейс метаданных AWS EC2 или токены Kubernetes PSAT). Затем сервер выдает узлу удостоверение в виде *x*509 SVID. Вот пример регистрации узла:

```
/opt/spire/bin/spire-server entry create \
  -spiffeID spiffe://production-trust-domain/nodes \
  -selector k8s_psat:cluster:production-cluster \
  -selector k8s_psat:agent_ns:spire \
  -selector k8s_psat:agent_sa:spire-agent \
  -node
```

Эта команда заставит сервер SPIRE назначить идентификатор SPIFFE `spiffe://production-trust-domain/nodes` всем узлам, на которых Pod агента соответствует заданным селекторам; в данном случае мы выбираем узлы с Pod'ами в пространстве имен SPIRE кластера `production-cluster` и запущенных от имени служебной SA `spire-agent` (проверенной с помощью PSAT).

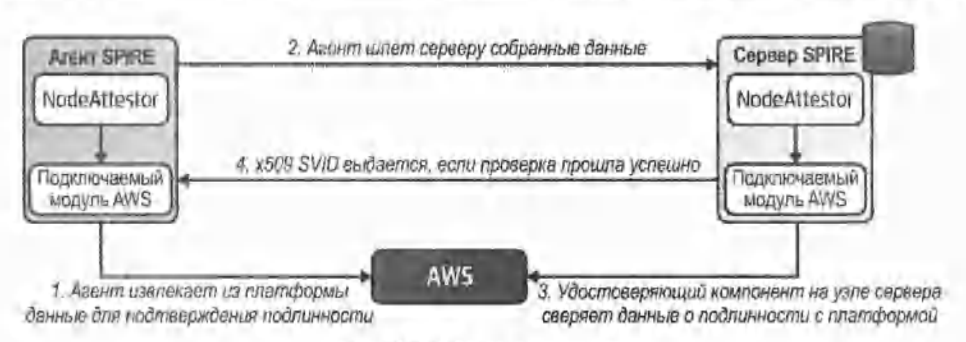


Рис. 10.13. Аттестация узла.
Рисунок позаимствован из официальной документации SPIRE (**https://oreil.ly/Q5eEW**)

После запуска приложения обращаются к API-интерфейсу своего узла для получения SVID. Агент SPIRE на основе информации, доступной ему на платформе (взятой из ядра, kubelet и т. д.), определяет свойства вызывающего приложения. Этот процесс называется *аттестацией рабочих заданий* (рис. 10.14). Затем сервер SPIRE сопоставляет эти свойства с учетными данными известных приложений на основе их селекторов и возвращает идентификатор SVID (через агента), который можно использовать для аутентификации в других системах:

```
/opt/spire/bin/spire-server entry create \
  -spiffeID spiffe://production-trust-domain/service-a \
  -parentID spiffe://production-trust-domain/nodes \
  -selector k8s:ns:default \
  -selector k8s:sa:service-a \
  -selector k8s:pod-label:app:frontend \
  -selector k8s:container-image:docker.io/johnharris85/service-a:v0.0.1
```

Эта команда заставляет сервер SPIRE назначить идентификатор SPIFFE `spiffe://production-trust-domain/service-a` всем приложениям, которые:

- выполняются на узле с идентификатором `spiffe://production-trust-domain/nodes`;
- находятся в пространстве имен `default`;
- запущены от имени служебной учетной записи `service-a`;
- имеют Pod с меткой `app: frontend`;
- были собраны из образа `docker.io/johnharris85/service-a:v0.0.1`.



Обратите внимание на то, что подключаемый модуль WorkloadAttestor может обращаться к kubelet (за информацией о приложениях), используя свою служебную учетную запись. Затем kubelet через API-интерфейс `TokenReview` проверяет токены носителя. Это требует наличия доступа к API-серверу Kubernetes. Следовательно, если API-сервер выйдет из строя, аттестация приложений может нарушиться.

Флаг `--authentication-token-webhook-cache-ttl` определяет, насколько долго ответы TokenReview должны храниться в кэше kubelet, и может помочь с минимизацией этой проблемы. Однако большие значения TTL кэша задавать не рекомендуется, так как это может сказаться на процессе отзыва прав доступа. Подробности можно найти в документации WorkloadAttestor SPIRE (**https://oreil.ly/Pn1ZP**).

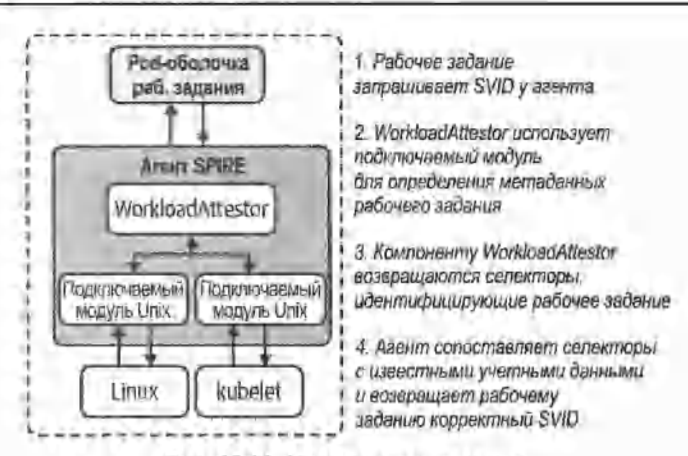


Рис. 10.14. Аттестация приложения.
Рисунок позаимствован из официальной документации SPIRE (**https://oreil.ly/Eh7XI**)

Методика, описанная в этом разделе, дает существенные преимущества при создании надежной системы идентификации приложений как на платформе Kubernetes, так и за ее пределами. В спецификации SPIFFE применяются общепринятые и широко поддерживаемые криптографические стандарты из *x*509 и JWT, а реализация SPIRE поддерживает множество разных способов интеграции приложений. Еще одно ключевое свойство — возможность очень точно определить область действия удостоверения за счет сочетания проецируемых токенов служебных учетных записей и собственных селекторов этого механизма, что позволяет идентифицировать отдельные Pod'ы. Это может быть особенно полезным в ситуациях, когда в Pod'е присутствуют sidecar контейнеры, и каждому из них нужен свой уровень доступа.

Описанный подход также, несомненно, наиболее трудозатратный, поскольку он требует умения работы с инструментарием и усилий на обслуживание еще одного компонента в окружении. Регистрация каждого приложения тоже может потребовать усилий, хотя этот процесс можно автоматизировать (и сообщество уже ведет работы в этом направлении).

У SPIFFE/SPIRE есть ряд механизмов интеграции с приложениями. То, какой из них лучше выбрать, зависит от желаемого уровня связывания с платформой и степени контроля над окружением, который имеют пользователи.

Прямой доступ из приложений

SPIRE предоставляет пакеты разработки для Go, C и Java, с помощью которых приложения могут напрямую интегрироваться с API-интерфейсом приложений SPIFFE. Эти пакеты являются обертками вокруг существующих HTTP-библиотек, но обеспечивают встроенную поддержку получения и проверки удостоверений. В примере на языке Go, показанном далее, выполняется вызов к Сервису Kubernetes `service-b`, в ответ на который должен быть предъявлен идентификатор SPIFFE (через *x*509 SVID):

```
err := os.Setenv("SPIFFE_ENDPOINT_SOCKET",
  "unix:///run/spire/sockets/agent.sock")
```

```
conn, err := spiffe.DialTLS(ctx, "tcp", "service-b",
  spiffe.ExpectPeer("spiffe://production-trust-domain/service-b"))
if err != nil {
  log.Fatalf("Unable to create TLS connection: %v", err)
}
```

Агент SPIRE также предоставляет доступ к интерфейсу gRPC (**https://grpc.io/about**), рассчитанному на пользователей, которые хотят более тесной интеграции с платформой, но применяют язык, для которого нет пакета разработки.

Прямая интеграция, описанная в этом подразделе, *не* рекомендуется для приложений, взаимодействующих с конечными пользователями, и для этого есть несколько причин:

- она тесно привязывает приложение к платформе/реализации;
- требует подключения Unix-сокета, принадлежащего агенту SPIRE, к Pod'у;
- ее сложно расширять.

Основная область, где использование этих библиотек напрямую себя оправдывает, — разработка каких-то вспомогательных инструментов для работы с платформой, которые служат оберткой или расширением существующих возможностей.

Прицепной прокси-сервер

SPIRE имеет встроенную поддержку Envoy SDS API для публикации сертификатов, предназначенных для прокси-сервера Envoy. С их помощью Envoy может устанавливать TLS-соединения с другими Сервисами и проверять входящие соединения посредством так называемого пакета доверия (англ. trust bundle).

Прокси-сервер Envoy можно настроить так, чтобы к нему могли подключаться только обладатели определенных идентификаторов SPIFFE (закодированных в SVID). Это можно реализовать двумя способами:

- Указать список значений `verify_subject_alt_name` в конфигурации Envoy.
- Использовать API-интерфейс Envoy для внешней авторизации, чтобы делегировать принятие решений о допуске внешней системе (например, Open Policy Agent).

В листинге 10.22 показана политика Rego, которая позволяет этого добиться.

Листинг 10.22

```
package envoy.authz

import input.attributes.request.http as http_request
import input.attributes.source.address as source_address

default allow = false

allow {
    http_request.path == "/api"
    http_request.method == "GET"
```

```
    svc_spiffe_id == "spiffe://production-trust-domain/frontend"
}

svc_spiffe_id = client_id {
    [_, _, uri_type_san] := split(
      http_request.headers["x-forwarded-client-cert"], ";")
    [_, client_id] := split(uri_type_san, "=")
}
```

В приведенном примере Envoy сопоставляет TLS-сертификат запроса с пакетом доверия SPIRE и затем делегирует авторизацию системе Open Policy Agent (OPA). Политика Rego анализирует SVID и пропускает запрос дальше, если идентификатор SPIFFE совпадает с `spiffe://production-trust-domain/frontend`. Архитектура этого процесса показана на рис. 10.15.



В рамках этого подхода система OPA внедряется в важный процесс обработки запросов, и это необходимо учитывать при проектировании процесса/архитектуры.

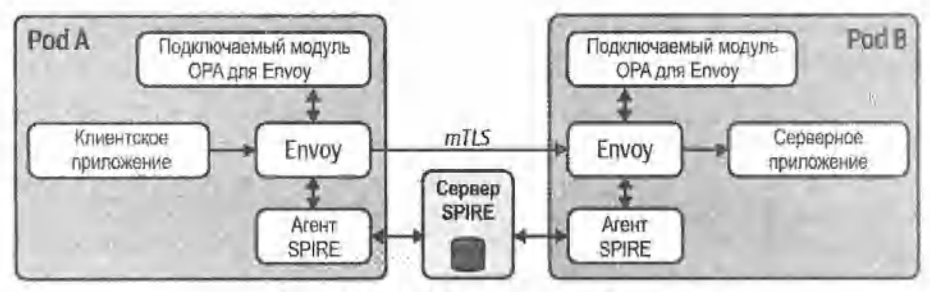


Рис. 10.15. SPIRE в сочетании с Envoy

Mesh-сеть (Istio)

Центр сертификации Istio создает SVID для всех служебных учетных записей, кодируя идентификатор SPIFFE в формате

`spiffe://cluster.local/ns/<пространство_имен>/sa/<служебная_учетная_запись>`

Благодаря этому Сервисы в mesh-сети Istio могут использовать конечные точки с поддержкой SPIFFE.



Несмотря на то, что mesh-сети выходят за рамки этой главы, многие из них пытаются взять на себя идентификацию и аутентификацию. Зачастую это подразумевает применение или расширение методов и инструментов, описанных в этой главе.

Другие методы интеграции приложений

Помимо основных методов, которые мы только что обсудили, SPIRE поддерживает следующие:

- Извлечение SVID и пакетов доверия непосредственно из файловой системы. Это позволяет приложениям распознавать изменения и перезагружаться, благодаря

чему их можно сделать менее ориентированными на SPIRE. С другой стороны, возникает риск похищения сертификатов, хранящихся в файловой системе.

♦ Использование модуля Nginx, который обеспечивает возможность доставать сертификаты из SPIRE в виде потока (похоже на интеграцию с Envoy, описанную ранее). Для Nginx существуют пользовательские модули, позволяющие указывать идентификаторы SPIFFE, которым должно быть позволено подключаться к серверу.

Интеграция с хранилищем конфиденциальных данных (Vault)

SPIRE подойдет для решения проблемы безопасного представления, если приложению нужно получить какие-то общие конфиденциальные данные из HashiCorp Vault (**https://www.vaultproject.io**). Vault можно сконфигурировать для аутентификации клиентов с использованием федерации OIDC на основе сервера SPIRE и провайдера OIDC.

Роли в Vault можно привязывать к определенным темам Vault (идентификаторам SPIFFE). В результате, когда приложение запрашивает JWT SVID из SPIRE, этого было достаточно для получения роли и, следовательно, учетных данных для доступа к Vault.

Интеграция с AWS

SPIRE также можно применять для идентификации и аутентификации в сервисах AWS. Этот процесс основан на той же идее с федерацией OIDC, которая была представлена в разделах о AWS IRSA и Vault. Приложение запрашивает JWT SWID, которые затем проверяются платформой AWS путем сопоставления с федеративным провайдером OIDC (сервером SPIRE). Недостаток этого подхода в том, что для обнаружения набора ключей JWKS (JSON Web Key Set), необходимых для проверки токенов JWT, платформе AWS нужен открытый доступ к серверу SPIRE.

Резюме

В этой главе мы подробно обсудили методы и инструменты, встречающиеся нам в реальных условиях, которые мы сами успешно применяем.

Тему идентификации можно рассматривать на разных уровнях, и ваш подход будет со временем меняться в зависимости от того, насколько уверенно вы себя чувствуете при работе с методами разной степени сложности, и насколько они соответствуют требованиям каждой отдельной организации. Если речь идет об идентификации пользователей, у нас обычно уже есть какая-то сторонняя система единого входа, но ее интеграция непосредственно в Kubernetes с помощью OIDC может оказаться нетривиальной задачей. В таких ситуациях мы видели, как организации выносят Kubernetes *за пределы* своей основной стратегии идентификации. Это может быть нормально в зависимости от требований, но прямая интеграция даст вам куда больше

прозрачности и контроля за окружениями, особенно теми, которые содержат больше одного кластера.

Если говорить о приложениях, то мы часто видим, что к идентификации (выходящей за рамки служебных учетных записей по умолчанию) относятся как к чему-то второстепенному. Опять же, в зависимости от внутренних требований это может быть нормально. Нельзя отрицать, что разработка надежной системы идентификации приложений как внутри кластера, так и между платформами, (в некоторых случаях) существенно усложняет задачу и требует более глубокого знания внешних инструментов. Тем не менее, когда организация достигает определенного уровня интеграции с Kubernetes, реализация методов, описанных в этой главе, как нам кажется, может серьезно усилить механизмы безопасности в ваших окружениях и обеспечить дополнительную степень глубинной защиты на случай взлома.

ГЛАВА 11

Создание сервисов платформы

Сервисы платформы приложений — это те компоненты, которые устанавливаются для расширения ее возможностей. Их обычно развертывают в пространстве имен вида `*-system` в качестве контейнерных приложений, а их обслуживанием занимается группа инженеров платформы. Эти сервисы отличаются от приложений, которые администрируют пользователи платформы, последние обслуживаются разработчиками приложений.

Облачно-ориентированная экосистема полна проектов, которые вы можете использовать как часть своей платформы приложений. К тому же многие поставщики рады предложить для вашей платформы свои решения. Задействуйте их, если они оправдывают себя с точки зрения затрат и преимуществ. На их основе можно даже сформировать всю платформу приложений. Но, как показывает наш опыт, корпоративные пользователи платформ, основанных на Kubernetes, предпочитают создавать собственные компоненты. У вас может возникнуть необходимость интегрировать свою платформу с какой-то существующей внутренней системой. К вашим приложениям могут предъявляться какие-то уникальные, замысловатые требования. Возможно, вам нужно учитывать крайние случаи, которые являются редкостью или обусловлены вашими конкретными бизнес-потребностями. Независимо от обстоятельств, в данной главе подробно описывается процесс расширения платформы приложений с применением собственных решений, призванных заполнить эти пробелы.

Основополагающий аспект разработки сервисов платформы — стремление избавиться от человеческого труда. И речь не только об автоматизации. Если автоматизация служит "краеугольным камнем", то интеграцию автоматизированных компонентов можно считать "строительным раствором". Налаженное, надежное взаимодействие между системами — сложная, но в то же время крайне важная цель. Идея программного обеспечения, ориентированного на API-интерфейсы, открывает широкие возможности, способствуя интеграции программных систем. Это одна из причин столь широкого распространения проекта Kubernetes: он позволяет сделать поведение всей платформы API-ориентированным, не требуя создания и предоставления API-интерфейсов для каждого программного компонента, который вы в нее добавляете. Компоненты могут пользоваться API-интерфейсом Kubernetes, работая с его встроенными ресурсами или добавляя свои собственные, которые представляют состояние новых объектов. Если следовать этим принципам интеграции при разработке сервисов платформы, то можно в значительной степени сократить человеческий труд. И, если мы добьемся успеха на этом поприще, перед нами откроются более широкие возможности для инноваций, развития и прогресса.

В данной главе речь пойдет о методах расширения плоскости управления Kubernetes. Мы берем эффективные инженерные подходы, применяемые проектом Kubernetes, и создаем платформы на этой основе. Мы посвятим бóльшую часть главы исследованию операторов Kubernetes, их шаблонов проектирования и сценариев использования, а также тому, как они разрабатываются. Но сначала необходимо провести краткий обзор механизмов расширения Kubernetes, чтобы получить целостное представление о создании сервисов платформы. Мы должны иметь четкий контекст и применять решения, гармонирующие с системой в целом. В конце будет рассмотрен процесс расширения, наверное, самого важного контроллера в экосистеме Kubernetes: планировщика.

Механизмы расширения

Kubernetes — это замечательная расширяемая система. Расширяемость, несомненно, является одной из ее мощнейших возможностей. Разработчики программного обеспечения часто допускают фатальную ошибку, пытаясь покрыть каждый потенциальный рабочий сценарий. Система может быстро превратиться в лабиринт из возможных вариантов выполнения с неясными путями достижения результатов. Более того, она зачастую становится неустойчивой по мере разрастания внутренних зависимостей и ухудшения стабильности из-за хрупких связей между ее компонентами. У центрального принципа философии Unix, согласно которому программы должны делать что-то одно, делать это хорошо и уметь взаимодействовать между собой, есть весомое обоснование. Сам по себе проект Kubernetes никогда бы не смог удовлетворить каждое возможное требование, возникающее у пользователей, которые оркестрируют свои контейнерные приложения. Такую идеальную систему создать невозможно. Одни только встроенные функции, которые предоставляет Kubernetes, вызывают достаточно затруднений. Поэтому, несмотря на довольно узкий круг обязанностей, Kubernetes в своем нынешнем виде является относительно сложной распределенной системой. Она никогда бы не смогла удовлетворить все требования, и ей не нужно этого делать, так как у нее есть механизмы расширения, которые позволяют удовлетворять специализированные нужды с помощью дополнительных решений с возможностью легкой интеграции. Kubernetes можно расширять и настраивать для выполнения практически любых ваших задач.

Большая часть информации о подключаемых расширениях, удовлетворяющих определенным интерфейсам для взаимодействия с Kubernetes, а также о популярных решениях на основе веб-хуков, разбросана по разным главам этой книги. Здесь мы сначала дадим краткий обзор, чтобы подготовить почву для дальнейшего рассмотрения темы операторов, которой посвящена значительная часть этой главы.

Подключаемые расширения

К ним относится широкая категория модулей, которые обычно помогают интегрировать Kubernetes со смежными системами, играющими важную, и нередко ключевую, роль в выполнении приложений. Это не готовые компоненты, а, скорее, спе-

цификации, которыми могут пользоваться сторонние разработчики для реализации своих решений:

- *Сеть* — CNI (Container Networking Interface — интерфейс управления сетью контейнеров) определяет интерфейс, который должен быть удовлетворен подключаемым модулем, объединяющим контейнеры в сеть. У этого интерфейса есть много реализаций, но все они должны ему соответствовать. Эта тема рассмотрена в *главе 5*.
- *Хранилище данных* — CSI (Container Storage Interface — интерфейс хранилищ для контейнеров) предоставляет контейнерным приложениям механизм для работы с системами хранения данных. Как и в предыдущем случае, существует множество подключаемых модулей, которые предоставляют доступ к хранилищам различных провайдеров. Данная тема рассмотрена в *главе 4*.
- *Среда выполнения контейнеров* — CRI (Container Runtime Interface — интерфейс среды выполнения контейнеров) определяет стандарт для операций, которые должна предоставлять среда выполнения контейнеров, чтобы агенту kubelet было все равно, какую среду вы используете. Самой распространенной реализацией традиционно является Docker, но в настоящее время набирают популярность другие альтернативы со своими достоинствами. Эта тема подробно обсуждается в *главе 3*.
- *Устройства* — система подключаемых устройств Kubernetes позволяет приложениям взаимодействовать с оборудованием своих узлов. Как показывает наш опыт, самым распространенным примером этого являются графические адаптеры (англ. Graphics Processing Units или GPUs), которые необходимы для ресурсоемких приложений. Узлы с такими специализированными устройствами нередко объединяются в пулы, на которых можно развернуть приложения. Подробней об этом можно почитать в *главе 2*.

Разработкой подключаемых модулей обычно занимаются поставщики, которые либо поддерживают, либо продают интегрируемые продукты. Судя по нашему опыту, очень немногие разработчики платформ создают собственные решения в этой области. Обычно все сводится к анализу доступных вариантов и выбору тех из них, которые удовлетворяют вашим требованиям.

Расширения на основе веб-хуков

Расширения на основе веб-хуков выступают внутренним компонентом, к которому обращается API-сервер Kubernetes для выполнения видоизмененных версий встроенных в API-интерфейс функций. Любой запрос, пришедший к API-серверу, проходит через несколько этапов. Клиент аутентифицируется, чтобы подтвердить наличие доступа (AuthN). API-сервер проверяет, авторизован ли клиент для выполнения действия, которое он запрашивает (AuthZ). API-сервер модифицирует ресурс в соответствии с инструкциями, полученными от активных подключаемых модулей допуска. Проверяется структура ресурса, проверяющий контроллер допуска выполняет любые специализированные или дополнительные проверки. На рис. 11.1

показано, как связаны между собой клиенты, API-интерфейс Kubernetes и используемые им расширения на основе веб-хуков.

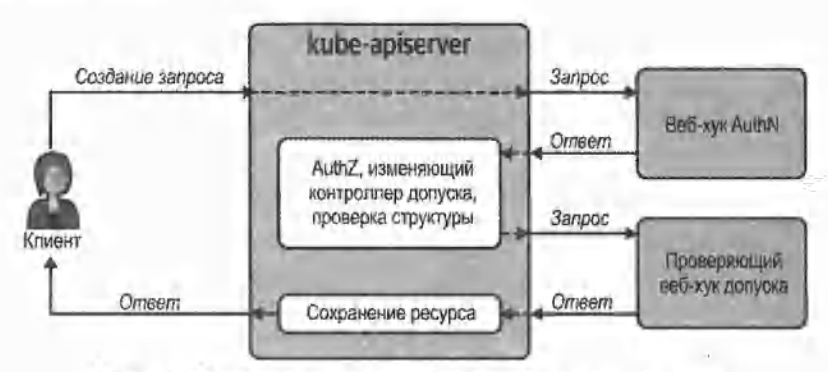


Рис. 11.1. Расширения на основе веб-хуков — это компоненты, к которым обращается API-сервер Kubernetes

Расширения для аутентификации

Расширения для аутентификации такие, как OpenID Connect (OIDC), дают возможность аутентифицировать запросы, направленные к API-серверу, с помощью внешней системы. Эта тема подробно рассмотрена в *главе 10*.

Вы также можете сделать так, чтобы API-сервер обращался к веб-хуку для авторизации действий, которые могут выполнять с ресурсом аутентифицированные пользователи. Такая реализация встречается нечасто, поскольку в Kubernetes встроена мощная система управления доступом на основе ролей. Тем не менее, если она вам по какой-либо причине не подходит, вы всегда можете применить данный подход.

Управление допуском

Управление допуском — это особенно полезный и широко используемый механизм расширения. При получении запроса на выполнение какого-то действия API-сервер обращается ко всем веб-хукам допуска, перечисленным в конфигурации проверяющих и изменяющих веб-хуков. Эта тема рассматривается в *главе 8*.

Операторы

Операторы — это клиенты API-сервера (а не веб-хуки для него, как в предыдущем случае). На рис. 11.2 видно, что они обращаются к API-интерфейсу Kubernetes точно так же, как это делают живые клиенты. Они обычно называются *операторами Kubernetes* и создаются в соответствии с официально задокументированным шаблоном проектирования "оператор" (**https://oreil.ly/HLXtJ**). Их основное назначение состоит в том, чтобы взять на себя часть обязанностей администраторов и выполнять операции от их имени. Эти расширения следуют тем же принципам проектирования, что и основные компоненты плоскости управления Kubernetes. При разработке операторов в виде сервисов платформы относитесь к ним, как к пользовательским расширениям плоскости управления для своей платформы приложений.

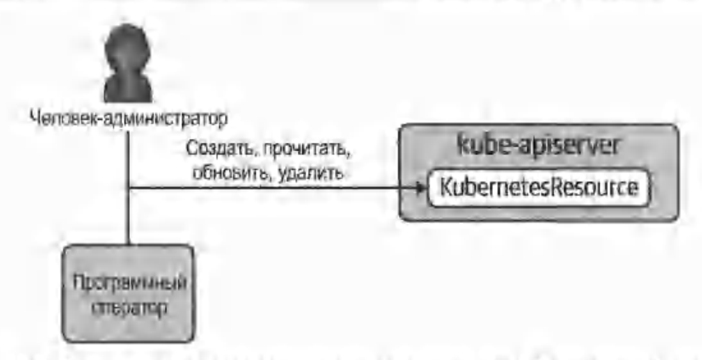


Рис. 11.2. Расширения-операторы являются клиентами API-сервера Kubernetes

Шаблон проектирования "оператор"

Можно сказать, что шаблон проектирования "оператор" сводится к расширению Kubernetes с помощью Kubernetes. Мы создаем новые ресурсы и контроллеры Kubernetes, чтобы согласовать состояние, которое в них определено. Для определения новых ресурсов используются CRD (Custom Resource Definitions — определения пользовательских ресурсов). CRD создают новые типы и описывают, как API-интерфейс Kubernetes должен их проверять. Теперь мы берем те же принципы и архитектурные решения, которые делают контроллеры Kubernetes такими эффективными, и применяем их для разработки программных расширений системы. Операторы создают с помощью двух основных механизмов: пользовательских ресурсов и контроллеров.

Понятие оператора было предложено в ноябре 2016 года Брэндоном Филлипсом, одним из основателей CoreOS. Изначально под оператором понимали контроллер для управления определенным сложным приложением, которое хранит свое состояние. Это определение по-прежнему актуально, но с годами оно несколько расширилось, и теперь, согласно документации Kubernetes, оператором считается любой контроллер, который использует CRD. Именно этот смысл мы вкладываем в понятие "оператор" в контексте создания сервисов платформы. Даже сервисы, которые не являются "сложными приложениями, хранящими свое состояние", могут выиграть от применения этого мощного шаблона проектирования.

В следующем разделе мы рассмотрим контроллеры Kubernetes, предоставляющие модель для тех возможностей, которые мы будем применять в наших собственных контроллерах. Вслед за этим мы исследуем пользовательские ресурсы, хранящие желаемое и текущее состояния, которые наши контроллеры будут для нас согласовывать.

Контролеры Kubernetes

Контроллеры реализуют основные возможности и функции Kubernetes. Они следят за разными типами ресурсов и реагируют на их создание, изменение и удаление,

приводя их в желаемое состояние. Например, в состав kube-controller-manager входит контроллер, который следит за ресурсами типа ReplicaSet. В момент создания одного из таких ресурсов он генерирует ряд идентичных Pod'ов — ровно столько, сколько было указано в поле `replicas` в спецификации ReplicaSet. Позже, если значение этого поля поменяется, контроллер создаст или удалит Pod'ы, чтобы достичь нового желаемого состояния.

Подобный механизм слежения лежит в основе возможностей всех контроллеров Kubernetes. Это функция etcd, которую API-сервер предоставляет контроллерам, реагирующим на изменения в ресурсах. Контроллеры поддерживают открытое соединение с API-сервером, благодаря чему последний может уведомлять их об обновлении ресурсов, которые их интересуют или которыми они управляют.

В результате можно реализовать очень сложное поведение. Пользователь может отправлять манифесты ресурсов для объявления состояния, к которому он хочет привести систему. Контроллеры, ответственные за достижение желаемого состояния, получают уведомления и начинают приводить текущее состояние к желаемому. Манифесты могут подаваться не только пользователями, но и самими контроллерами, инициируя тем самым работу других контроллеров. В итоге получается система, способная предоставлять сложные, но при этом стабильные и надежные возможности.

У этих контроллеров есть одна важная особенность: если ввиду каких-то трудностей им не удается достичь желаемого состояния, они повторяют свои попытки в бесконечном цикле. Попытки никогда не прекращаются, хотя интервал между ними может со временем увеличиться, чтобы не слишком нагружать систему. В результате получается механизм самовосстановления, который чрезвычайно важен в сложных распределенных системах, подобных этой.

Например, планировщик отвечает за распределение Pod'ов по узлам кластера. Но это все лишь очередной контроллер, просто с особенно важной и сложной обязанностью. Если одной или нескольким Pod'ам не хватает вычислительных ресурсов, они войдут в состояние ожидания ("Pending"), и планировщик продолжит периодически пытаться их развернуть. Таким образом, если в какой-то момент вычислительные ресурсы освободятся или расширятся, Pod'ы будут развернуты и запущены. Поэтому, если какое-то другое приложение с пакетной обработкой завершится и освободит ресурсы, или если контроллер автомасштабирования добавит в кластер новые рабочие узлы, ожидающий Pod будет развернут без вмешательства со стороны администратора.

Если вы создаете расширения для своей платформы приложений с помощью шаблона проектирования "оператор", вам необходимо руководствоваться принципами, которые присущи контроллерам Kubernetes:

- следите за ресурсами в API-интерфейсе Kubernetes, чтобы получать уведомления об изменении их желаемого состояния;
- пытайтесь согласовать текущее и желаемое состояния в непрекращающемся цикле.

Пользовательские ресурсы

Одна из важнейших особенностей API-интерфейса Kubernetes — возможность расширять типы ресурсов, которые он распознает. Применив корректный CRD (англ. Custom Resource Definition), вы немедленно получите в свое распоряжение новый пользовательский тип API. CRD содержит все поля, которые должны присутствовать в вашем пользовательском ресурсе, как в разделе `spec`, где описывается его желаемое состояние, так и в разделе `status`, где можно записывать важную информацию о наблюдаемом, текущем состоянии.

Прежде чем углубляться в эту тему, давайте проведем краткий обзор ресурсов Kubernetes. Здесь и далее им уделено много внимания, поэтому необходимо убедиться в том, что у нас с вами нет никаких разночтений. В Kubernetes под *ресурсами* понимают объекты, которые служат для записи состояния. Примером распространенного ресурса является Pod. При создании его манифеста вы определяете атрибуты того, что впоследствии станет ресурсом `Pod`. Передав его API-серверу с помощью `kubectl apply -f pod.yaml` или похожей команды, вы создадите экземпляр типа `Pod`. С одной стороны, мы имеем тип API, который обозначает определение и вид объекта в соответствии с его описанием в CRD. С другой, мы получаем ресурс, который представляет собой реализацию или экземпляр этого типа. `Pod` — это тип API. Pod с именем "my-app", который вы создаете, — это ресурс Kubernetes.

В отличие от реляционной базы данных, где отношения между объектами записываются и связываются с помощью внешних ключей, хранящихся в самой БД, каждый объект API-интерфейса Kubernetes независим. Отношения формируются за счет наличия меток и селекторов, а их управлением занимаются контроллеры. Вы не можете запросить у etcd связанные объекты, как это делается в структурированном языке запросов (англ. Structured Query Language или SQL). Поэтому, когда обсуждаются ресурсы, речь идет об экземплярах объектов `Namespace`, `Pod`, `Deployment`, `Secret`, `ConfigMap` и т. д. А пользовательскими называют ресурсы, определенные пользователями и добавленные в виде CRD. Создавая CRD, вы определяете новый тип API, с помощью которого пользовательские ресурсы можно создавать и администрировать точно так же, как и те, что входят в ядро Kubernetes.

Спецификация Open API v3, предназначенная для определения полей CRD, дает такие возможности, как задание необязательных и обязательных полей, а также указание значений по умолчанию. Это создает инструкции для API-сервера, с помощью которых он проверяет полученный запрос и создает или обновляет один из ваших пользовательских ресурсов. Кроме того, вы можете группировать свои API-интерфейсы для улучшения их логической организации и, что немаловажно, управлять версиями своих типов API.

Чтобы продемонстрировать, что такое CRD и как выглядит манифест для итогового пользовательского ресурса, рассмотрим гипотетический пример с пользовательским типом API WebApp. В этом примере ресурс WebApp содержит желаемое состояние приложения, в состав которого входят следующие шесть ресурсов Kubernetes:

- *Deployment* — приложение, которое не хранит свое состояние и предоставляет клиентам пользовательский интерфейс, обрабатывает запросы и записывает данные в реляционную БД.

- *StatefulSet* — реляционная БД, которая отвечает за постоянное хранение данных для веб-приложения.
- *ConfigMap* — содержит конфигурационный файл приложения, который подключается к каждому Pod'у объекта Deployment.
- *Secret* — содержит учетные данные, с помощью которых приложение подключается к своей БД.
- *Ingress* — содержит правила маршрутизации для контроллера Ingress, с помощью которых тот маршрутизирует клиентские запросы в кластере.

Создание ресурса `WebApp` заставит оператора WebApp сгенерировать все эти дочерние ресурсы, которые сформируют полноценный, рабочий экземпляр веб-приложения, обслуживающий конечных пользователей и бизнес-клиентов.

В листинге 11.1 показано, как может выглядеть CRD для определения нового типа API WebApp.

Листинг 11.1. Манифест пользовательского ресурса WebApp

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: webapps.workloads.apps.acme.com ❶
spec:
  group: workloads.apps.acme.com
  names: ❷
    kind: WebApp
    listKind: WebAppList
    plural: webapps
    singular: webapp
  scope: Namespaced
  versions:
  - name: v1alpha1
    schema:
      openAPIV3Schema:
        description: WebApp is the Schema for the webapps API
        properties:
          apiVersion:
            description: 'APIVersion defines the versioned schema of this
              representation of an object. Servers should convert recognized
              schemas to the latest internal value, and may reject unrecognized
              values.'
            type: string
          kind:
            description: 'Kind is a string value representing the REST resource
this
              object represents. Servers may infer this from the endpoint the
client
```

```
            submits requests to. Cannot be updated. In CamelCase.'
          type: string
        metadata:
          type: object
        spec:
          description: WebAppSpec defines the desired state of WebApp
          properties:
            deploymentTier: ❸
              enum:
              - dev
              - stg
              - prod
              type: string
            webAppHostname:
              type: string
            webAppImage:
              type: string
            webAppReplicas: ❹
              default: 2
              type: integer
          required: ❺
          - deploymentTier
          - webAppHostname
          - webAppImage
          type: object
        status:
          description: WebAppStatus defines the observed state of WebApp
          properties:
            created:
              type: boolean
          type: object
      type: object
  served: true
  storage: true
```

❶ Имя *определения* пользовательского ресурса (не путать с именем самого ресурса).

❷ Различные вариации имени пользовательского ресурса, включая множественное число (не путать с именем определения).

❸ Поле `deploymentTier` должно содержать одно из значений, перечисленных в `enum`. API-сервер проверяет выполнение данного условия при получении запросов на создание или обновление экземпляра этого пользовательского ресурса.

❹ Поле `webAppReplicas` содержит значение по умолчанию, которое будет применено, если не предоставить собственное значение.

❺ Здесь перечислены обязательные поля. Обратите внимание на то, что здесь отсутствует поле `webAppReplicas`, так как у него есть значение по умолчанию.

Теперь давайте посмотрим, как бы выглядел манифест для WebApp. Прежде чем передавать API-серверу манифест, показанный в листинге 11.2, необходимо создать ресурс CRD из листинга 11.1, чтобы у Kubernetes был его API-интерфейс. В противном случае Kubernetes не поймет, что вы пытаетесь создать.

Листинг 11.2. Пример манифеста для ресурса WebApp

```
apiVersion: workloads.apps.acme.com/v1alpha1
kind: WebApp
metadata:
  name: webapp-sample
spec:
  webAppReplicas: 2 ❶
  webAppImage: registry.acme.com/app/app:v1.4
  webAppHostname: app.acme.com
  deploymentTier: dev ❷
```

❶ Этот манифест определяет значение по умолчанию для необязательного поля. Это нормально, если нужно внести ясность, хотя без этого можно обойтись.

❷ Используется одно из значений, допустимых для этого поля. Любое недопустимое значение заставит API-сервер отклонить запрос с ошибкой.

Когда манифест WebApp передается API-интерфейсу Kubernetes, оператор WebApp получает уведомление о создании нового экземпляра этого типа и приводит его в желаемое состояние, выраженное в манифесте. Для этого он обращается к API-серверу, чтобы создать различные дочерние ресурсы, необходимые для запуска экземпляра веб-приложения.



Несмотря на широкие возможности модели пользовательских ресурсов, не стоит ею злоупотреблять. Не используйте CRD в качестве основного хранилища данных для приложений, с которыми работают конечные пользователи. Kubernetes — это система оркестрации контейнеров. В etcd должно храниться состояние ваших развертываний, а не внутренние данные приложений, предназначенные для постоянного хранения, иначе возникает серьезная нагрузка на плоскость управления кластера. Применяйте реляционные базы данных, объектные хранилища или любые другие решения, которые подходят для вашего приложения. А развертываниями пусть занимается плоскость управления.

Сценарии использования операторов

Операторы предлагают хорошую модель для расширения платформы на основе Kubernetes. Если состояние системы, которую нужно реализовать, можно представить в виде полей пользовательского ресурса, и если вам подходит механизм согласования изменений с помощью контроллера Kubernetes, то операторы зачастую являются отличным выбором.

Помимо расширения возможностей платформы, операторы также можно применять для облегчения администрирования развертываний ПО на вашей платформе. Они могут иметь вид удобной обобщенной абстракции, но вы также можете адаптиро-

вать их под нужды конкретного сложного приложения. В любом случае программное управление развертыванием программного обеспечения — очень полезный подход.

В этом разделе мы обсудим три основные категории операторов, которые вы можете использовать в своей платформе:

- служебные компоненты платформы;
- операторы приложений общего назначения;
- операторы для отдельно взятых приложений.

Служебные компоненты платформы

Операторы могут быть чрезвычайно полезными для развития вашей платформы. Они позволяют добавлять возможности в кластер и расширять функциональность Kubernetes за счет прозрачного функционирования плоскости управления и интеграции с ней. Существует множество готовых проектов с открытым исходным кодом, которые предоставляют сервисы платформы поверх Kubernetes с помощью операторов. И хотя эта глава посвящена созданию собственных компонентов, готовые проекты здесь упоминаются неспроста. Наблюдая за тем, как они работают, вы можете сформировать хорошую модель мышления. Если вы окажетесь в ситуации, в которой необходимо разработать собственные служебные компоненты платформы, вам будет полезно изучить существующие успешные проекты:

- Prometheus Operator (**https://oreil.ly/ClgDL**) дает вам возможность предоставлять в рамках своей платформы сервисы для сбора и хранения метрик, а также для генерации оповещений. В *главе 9* мы подробно рассматриваем то, какую пользу можно извлечь из этого проекта.
- cert-manager (**https://cert-manager.io**) позволяет предоставлять сервисы для управления сертификатами. Он значительно уменьшает объем работы администраторов и снижает вероятность сбоев, предлагая услуги по созданию и продлению сертификатов x509.
- Rook (**https://rook.io**) — оператор хранения данных, который интегрируется с такими провайдерами, как Ceph (**https://ceph.io**) для управления блочными, объектными и файловыми хранилищами в виде сервиса.

Эти решения с открытым исходным кодом являются примерами того, что нам предлагает сообщество. Существуют также многочисленные поставщики, которые могут предоставлять и поддерживать аналогичные служебные компоненты платформы. Но, если подходящего решения нет, организации иногда создают собственные компоненты.

Распространенный пример нестандартного служебного компонента платформы, который встречается нам в нашей работе, — оператор Namespace. Как показывает наш опыт, у многих организаций есть стандартный набор ресурсов таких, как ResourceQuota, LimitRange и Role, которые создаются в каждом пространстве имен. И всю утомительную работу по созданию этих ресурсов в каждом пространстве

удобно переложить на контроллер. В одном из следующих разделов мы воспользуемся этой концепцией оператора Namespace в качестве примера, чтобы проиллюстрировать некоторые аспекты реализации операторов.

Операторы приложений общего назначения

Основная специализация и обязанность разработчиков приложений состоит в расширении возможностей своего программного обеспечения и повышении его стабильности. Речь не о написании документов YAML для развертывания в Kubernetes. Умение корректно определять лимиты и запросы, способность подключать тома ConfigMap или объекты `Secret` в виде переменных окружения, умение посредством селекторов меток связывать Сервисы и Pod'ы— все это не имеет отношения к расширению возможностей или повышению стабильности ПО.

В организации, выработавшей устоявшиеся процедуры для развернутых приложений, модель абстрагирования сложных операций с помощью механизмов общего назначения выглядит многообещающе. Это особенно касается организаций, которые внедрили у себя микросервисные архитектуры. В таких окружениях может существовать довольно много программных компонентов с разными возможностями, но с очень похожими методами развертывания.

Например, если у вашей компании есть множество приложений, состоящих из ресурсов `Deployment`, `Service` и `Ingress`, то значительную часть манифестов этих объектов можно абстрагировать с помощью оператора, в котором воплощены определенные закономерности. Сервисы всегда ссылаются на метки, назначенные Pod'ам и объектам `Deployment`. Ingress всегда ссылается на имя Сервиса. Соблюдение всех этих правил — тяжелый труд, который вполне можно делегировать оператору.

Операторы для отдельно взятых приложений

Этот тип операторов Kubernetes сочетает в себе то, что лежит в основе самого их понятия: пользовательские ресурсы и пользовательский контроллер Kubernetes для управления сложными приложениями, хранящими свое состояние. Они ориентированы на работу с определенным приложением. Распространенные примеры этого подхода — различные операторы баз данных, разрабатываемые сообществом. У нас есть операторы для Cassandra, Elasticsearch, MySQL, MariaDB, PostgreSQL, MongoDB и многих других БД. Они обычно отвечают за первоначальное развертывание и последующие эксплуатационные действия (обновление конфигурации, резервное копирование и переход на новую версию).

За последние несколько лет популярность набрали операторы для широко распространенных проектов, которые поддерживаются как коммерческими поставщиками, так и сообществом. В корпоративных приложениях данный подход по-прежнему находится в зачаточном состоянии. Если ваша организация разрабатывает и поддерживает сложное приложение, хранящее свое состояние и предназначенное для внутреннего использования, то создание оператора специально для него может быть хорошим решением. Подобный вариант, к примеру, заслуживает вни-

мация в случае, если ваша компания занимается обслуживанием веб-сайта электронной торговли, приложения для обработки транзакций или системы инвентаризации, которые предоставляют важные бизнес-функции. Это отличная возможность сократить объем ручного труда в процессах развертывания и последующего администрирования такого рода приложений, особенно если они развертываются в крупных масштабах и часто обновляются.

Сказанное не означает, что эти операторы, предназначенные для конкретных приложений, всегда подходят для управления вашими приложениями. В более простых случаях они, скорее всего, будут излишними. Разработать операторы, готовые к промышленному внедрению, не так-то просто, поэтому взвесьте все плюсы и минусы. Сколько времени у вас уходит на рутинные операции по управлению развертыванием и дальнейшему администрированию приложения? Будет ли стоимость инженерных ресурсов, необходимых для создания оператора, меньше в долгосрочной перспективе по сравнению с этим тяжелым трудом? Могут ли более простые инструменты вроде Helm или Kustomize обеспечить достаточный уровень автоматизации для облегчения вашей работы?

Разработка операторов

Разработка операторов Kubernetes (особенно полнофункциональных и рассчитанных на определенные приложения) — нетривиальная задача. Инженерные ресурсы, которые могут потребоваться для подготовки такого сложного проекта, могут быть значительными. Как и в случае с другими видами разработки ПО, работу в этой области следует начинать с чего-то более простого, чтобы познакомиться с полезными методами и удачными стратегиями. В данном разделе мы обсудим некоторые инструменты и стратегии проектирования, которые помогут сделать разработку операторов более эффективной и успешной.

Мы рассмотрим некоторые проекты, способные помочь в создании таких инструментов. Затем мы подробно разберем процесс проектирования и реализации такого рода программного обеспечения. Для иллюстрации идей и рекомендованных подходов будут предоставлены примеры программного кода.

Инструментарий для разработки операторов

Если у вас есть подходящий рабочий сценарий для создания собственного оператора, в этом вам могут помочь несколько проектов, разрабатываемых сообществом. Если вы или кто-то из ваших коллег имеет богатый опыт программирования на Go, знаком с библиотекой Kubernetes client-go и умеет создавать операторы Kubernetes, значит, у вас определенно есть возможность написать собственный оператор с нуля. Однако у всех операторов есть кое-что общее, и даже опытные разработчики операторов нередко пользуются инструментами для генерации шаблонного и служебного исходного кода. Это просто экономит время. Пакеты разработки ПО (англ. Software Development Kits или SDK) и платформы, которые вписываются в модель вашего проекта, могут принести пользу, но в то же время они могут причинять не-

удобства, если в них заложены принципы, идущие вразрез с вашей целью. Если ваш проект соответствует стандартной модели, состоящей в применении одного или нескольких пользовательских ресурсов для определения конфигурации и пользовательских контроллеров для реализации поведения, относящегося к этим объектам, то инструменты, обсуждаемые здесь, с высокой долей вероятности будут вам полезны.

Kubebuilder

Kubebuilder можно охарактеризовать как SDK для создания API-интерфейсов Kubernetes. Это меткое описание, но на практике все не совсем так, как можно было бы ожидать. Работа с Kubebuilder начинается с утилиты командной строки, с помощью которой генерируется каркас интерфейса. Она создает исходный код, файлы Dockerfile и Makefile, примеры манифестов Kubernetes — все, что вам нужно написать в любом из таких проектов. Это экономит уйму времени на начальной стадии разработки.

Kubebuilder также задействует набор инструментов из родственного проекта под названием controller-runtime. Исходный код, сгенерированный утилитой командной строки, уже содержит необходимые инструкции импорта и распространенные реализации. Это избавляет вас от написания большей части шаблонного кода, связанного с выполнением контроллера и взаимодействием с API-интерфейсом Kubernetes. Это также помогает подготовить разделяемые кэши и клиентские вызовы для эффективного обращения к API-серверу. Кэш позволяет вашему контроллеру получать объекты и составлять их списки, не связываясь с API-сервером при каждом запросе. В результате снижается нагрузка на API-сервер и ускоряется процесс согласования. Controller-runtime также предоставляет механизмы для инициирования запросов на согласование в ответ на события, такие как изменение ресурса. Эти запросы инициируются по умолчанию для родительского пользовательского ресурса. Они также могут (и обычно должны) инициироваться при внесении изменений в дочерние ресурсы, созданные контроллерами и функциями, доступными для такого рода операций. Если ваш контроллер работает в режиме высокой доступности (англ. Highly Available или HA), то controller-runtime дает возможность включить механизм выбора лидера, чтобы в любой момент времени активным был только один контроллер. Более того, в состав controller-runtime входит пакет для реализации веб-хуков, которые зачастую управляют допуском. Наконец, эта библиотека включает в себя средства для ведения структурированного журнала и обеспечения наблюдаемости за счет открытия доступа к метрикам Prometheus.

Если вы программист на Go с опытом работы с Kubernetes, то Kubebuilder будет хорошим выбором. Данный инструмент подойдет даже опытным разработчикам ПО, которые не знакомы с языком программирования Go. Тем не менее, он предназначен только для Go и не ориентирован на другие языки.



Если вы собираетесь разрабатывать инструменты для Kubernetes, то мы настоятельно рекомендуем изучить Go (если вы этого еще не сделали). Конечно, для этого подойдут и другие языки. Kubernetes, в конце концов, предлагает REST API. К тому же, существуют официально поддерживаемые библиотеки для Python, Java,

> C#, JavaScript и Haskell, не говоря уже о многих других библиотеках, разработкой которых занимается сообщество. Если у вас есть веские причины для выбора вами одного из этих языков, ничто не должно помешать вам успешно это сделать. Однако сама платформа Kubernetes написана на Go, и для этого языка существует богатая и хорошо поддерживаемая экосистема.

Одна из особенностей проекта Kubebuilder, существенно экономящая наше время, состоит в том, что он генерирует пользовательские ресурсы. Написание манифестов CRD вручную — занятие не из простых. Спецификация OpenAPI v3, предназначенная для определения таких пользовательских API-интерфейсов, довольно развернутая и сложная. Утилита командной строки Kubebuilder генерирует файлы, в которых вы можете указать поля для своих пользовательских типов API. Вы добавляете различные поля в определения структур и затем помечаете их специальными маркерами, которые предоставляют такие метаданные как значения по умолчанию. После этого вы можете воспользоваться командой `make`, чтобы сгенерировать манифесты CRD. Это очень удобно.

Раз уж мы заговорили о команде `make`, то следует упомянуть, что она позволяет генерировать не только CRD, но и RBAC. С ее помощью также можно делать выборку манифестов пользовательских ресурсов, устанавливать пользовательские ресурсы в отладочном кластере, собирать и публиковать образы операторов и запускать контроллеры, работающие с кластером, локально на время разработки. Наличие удобных команд для всех этих утомительных, кропотливых задач, повышает продуктивность, особенно на ранних этапах разработки проекта.

В связи с этим мы предпочитаем и рекомендуем вам использовать Kubebuilder для создания операторов. Этот инструмент успешно применяется в целом ряде проектов.

Metacontroller

Если уверенно владеете любым другим языком, помимо Go, и предпочитаете именно его, то еще одним полезным инструментом в разработке операторов может послужить Metacontroller. Это совершенно отдельный подход к созданию и развертыванию операторов, но он заслуживает вашего внимания, если вы хотите сочетать разные языки программирования и собираетесь развертывать на своей платформе существенное количество операторов, написанных самостоятельно. Разработчики с опытом программирования для Kubernetes тоже иногда применяют Metacontroller для создания прототипов, после чего, определившись с архитектурой и деталями реализации, используют Kubebuilder для написания итогового проекта. И это говорит об одной из сильных сторон Metacontroller: после установки в кластере этого подключаемого модуля вы можете сразу же приступать к работе.

Таким образом, Metacontroller представляет собой дополнение к кластеру, которое инкапсулирует взаимодействие с API-интерфейсом Kubernetes. Ваша задача состоит в написании веб-хука, содержащего логику вашего контроллера. В терминологии Metacontroller он называется *лямбда-контроллером*. Ваш лямбда-контроллер принимает решения о том, что делать с ресурсами, которые его интересуют.

Metacontroller следит за доверенными ему ресурсами и, если в них вносятся изменения, требующие принятия какого-то решения, отправляет HTTP-вызов вашему

контроллеру. Сам Metacontroller использует CRD, которые определяют характеристики вашего веб-хука, например, его URL-адрес и ресурсы, которыми он управляет. Таким образом, в вашем кластере работает Metacontroller, процесс создания контроллера заключается в развертывании веб-хука и добавлении пользовательского ресурса Metacontroller (например, CompositeController). Вашему контроллеру остается только предоставить конечную точку, которая будет принимать запросы от Metacontroller, разбирать данные в формате JSON, содержащие экземпляр соответствующего ресурса Kubernetes, и возвращать обратно ответы с любыми изменениями, которые Metacontroller должен передать API-интерфейсу Kubernetes. На рис. 11.3 показано, как эти компоненты взаимодействуют при наличии Metacontroller.

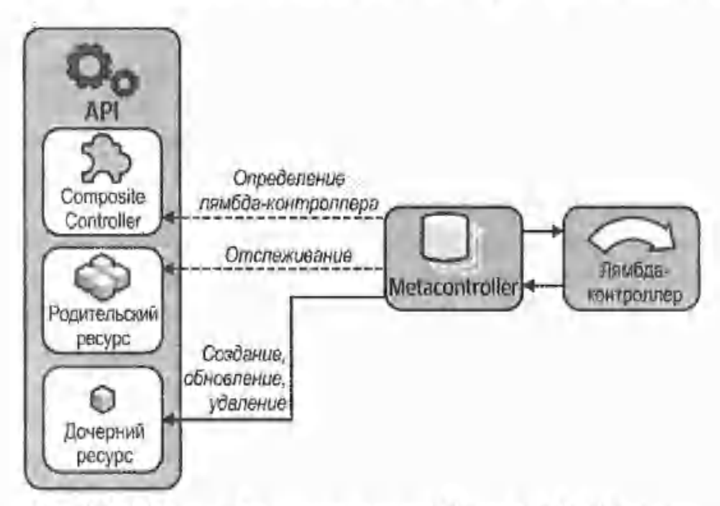


Рис. 11.3. Metacontroller инкапсулирует API-интерфейс Kubernetes для вашего лямбда-контроллера

С чем Metacontroller вам не поможет, так это с созданием любых CRD, которые нужно добавить в кластер. Это вам придется делать самостоятельно. Если вы пишете контроллер, реагирующий на изменения во встроенных ресурсах Kubernetes, затруднений не возникает. Но, если вы разрабатываете пользовательские ресурсы, Kubebuilder имеет в этой области существенное преимущество.

Operator Framework

Operator Framework — это набор инструментов с открытым исходным кодом, которые изначально были созданы в компании Red Hat, но теперь разрабатываются в рамках CNCF Incubator. Они помогают в создании операторов. В их состав входит пакет Operator SDK, который по своим возможностям похож на инструмент Kubebuilder, предназначенный для разработки операторов на Go. Как и у Kubebuilder, у него есть утилита командной строки для генерации шаблонного кода проекта. Еще одно сходство состоит в облегчении интеграции с API-интерфейсом Kubernetes за счет средств controller-runtime. Помимо Go, Operator SDK позволяет разработчикам использовать для выполнения операций Helm или Ansible. В состав Operator Framework также входит диспетчер Operator Lifecycle Manager, назначение

которого понятно из его названия: это оператор для ваших операторов. Он предоставляет абстракции для их установки и обновления. У него также есть веб-сайт, на котором пользователи могут искать операторы для своего программного обеспечения. На практике нам не встречались разработчики платформ, применяющие эти инструменты. Поддержкой этого проекта занимается Red Hat, поэтому он, вероятно, более популярен среди пользователей OpenShift — платформы на основе Kubernetes от той же компании.

Проектирование моделей данных

По аналогии с тем, как проектирование веб-приложения может начинаться с определения структуры БД, в которой оно будет хранить свои данные, отличным первым шагом в разработке оператора станет выбор модели данных для его пользовательских ресурсов. Еще до начала работы над пользовательским ресурсом у вас, вероятно, будет определенное представление о том, какие поля должны присутствовать в его спецификации. Как только вы определитесь с задачей, которую нужно решить, или пробелом, который нужно восполнить, атрибуты объекта, описывающего желаемое состояние, начнут обретать свою форму.

Оператор Namespace, который уже приводился в качестве примера в этой главе, может изначально создавать разнообразные ресурсы, такие как LimitRange, ResourceQuota, Role, NetworkPolicy, чтобы команда разработки приложения получила нужное ей пространство имен. Вы можете сразу привязать руководителя команды к роли `namespace-admin` и затем вверить ему управление этим пространством, в результате естественным образом прийдя к добавлению поля `adminUsername` в спецификацию пользовательского ресурса. Манифест такого ресурса может выглядеть так, как показано в листинге 11.3.

Листинг 11.3. Пример манифеста для ресурса AcmeNamespace

```
apiVersion: tenancy.acme.com/v1alapha1
kind: AcmeNamespace
metadata:
  name: team-x
spec:
  namespaceName: app-y ❶
  adminUsername: sam ❷
```

❶ Произвольное название для пространства имен. В данном случае в нем будет размещено приложение, "app-y".

❷ Это имя пользователя будет совпадать с тем, которое применяется провайдером идентификации компании. В роли последнего обычно выступает Active Directory или похожая система.

Применение манифеста из листинга 11.3 приведет к тому, что имя пользователя `sam` будет добавлено в список `subjects` объекта `RoleBinding`, относящегося к роли `namespace-admin`. В листинге 11.4 показано, как это происходит.

Листинг 11.4. Пример ресурсов Role и RoleBinding, созданных для пространства имен team-x типа AcmeNamespace

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: namespace-admin
  namespace: app-y
rules:
- apiGroups:
  - "*"
  resources:
  - "*"
  verbs:
  - "*"
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: namespace-admin
  namespace: app-y
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: namespace-admin
subjects:
- kind: User
  name: sam ❶
  namespace: app-y
```

❶ Значение `adminUsername`, предоставленное в манифесте AcmeNamespace, будет вставлено сюда и привязано к роли `namespace-admin`.

Если задуматься о том, чего мы хотим достичь (привязать пользователя `sam` к роли `namespace-admin`), становится довольно очевидно, какие данные нам для этого понадобятся: имя самого пользователя и название пространства имен. Поэтому давайте начнем с очевидной информации, необходимой для предоставления этой функциональности, и определим соответствующие поля в спецификации нашего CRD. В рамках проекта Kubernetes это будет выглядеть похоже на то, что показано в листинге 11.5.

Листинг 11.5. Определение типа для AcmeNamespaceSpec

```
// api/v1alpha1/acmenamespace_types.go

...

// AcmeNamespaceSpec определяет желаемое состояние AcmeNamespace
type AcmeNamespaceSpec struct {
```

```
    // Название пространства имен
    NamespaceName string `json:"namespaceName"`

    // Имя пользователя для namespace-admin
    AdminUsername string `json:"adminUsername"`
...
```

Это исходный код, на основе которого Kubebuilder сгенерирует манифест вашего пользовательского ресурса и выберет манифест AcmeNamespace, который будет использоваться для тестирования и в демонстрационных целях.

Итак, мы имеем модель данных, которая, как мы *считаем*, позволит нам адекватно управлять состоянием нужного нам механизма. Теперь пришло время написать контроллер. Вполне возможно, что по мере разработки и обнаружения дополнительных полей, которые нам понадобятся для получения желаемого результата, эта модель данных перестанет удовлетворять нашим требованиям. Но с чего-то нужно начинать.

Реализация бизнес-логики

Бизнес-логика реализуется в нашем контроллере, основная задача которого состоит в управлении одним или несколькими пользовательскими ресурсами. Контроллер отслеживает ресурсы, за которые он отвечает, и это довольно легко осуществить, если использовать такие инструменты, как Kubebuilder и Metacontroller, или даже просто библиотеку client-go (замечательные примеры кода, которые это демонстрируют, можно найти в ее репозитории на GitHub). Подписавшись на изменения своих пользовательских ресурсов, ваш контроллер будет получать уведомления при каждом их обновлении. На данном этапе его задача сводится к следующему:

- сформировать точную картину текущего состояния системы;
- проанализировать желаемое состояние системы;
- предпринять действия, необходимые для приведения текущего состояния к желаемому.

Текущее состояние

В сущности, существуют три места, из которых ваш контроллер может собирать информацию о текущем состоянии:

- поле `status` вашего пользовательского ресурса;
- другие ресурсы кластера, которые с ним связаны;
- условия, имеющие отношение к нашему ресурсу, но выходящие за рамки кластера или продиктованные другими системами.

Раздел `status` — это место, куда контроллеры Kubernetes могут записывать наблюдаемое, текущее состояние. Например, Kubernetes использует поле `status.phase` некоторых ресурсов, таких как Pod и Namespace, чтобы следить за тем, на какой ста-

дии выполнения они находятся: Running (в случае с Pod'ами) или Active (в случае с пространствами имен).

Давайте вернемся к примеру с оператором Namespace. Контроллер уведомляется о новом ресурсе AcmeNamespace и получает его спецификацию. Он не может исходить из того, что это новый ресурс, и просто автоматически создать дочерние объекты Namespace и Role. Возможно, этот ресурс уже существовал и просто был обновлен. В ответ на попытку повторного создания дочерних объектов API-интерфейс Kubernetes вернет ошибку. Но, согласно предыдущему примеру, если поместить в раздел status нашего CRD поле phase, то контроллер сможет его проверить и оценить его текущее состояние. Сразу после создания ресурса его поле status.phase будет пустым. Таким образом контроллер будет знать, что это новый ресурс, и сможет приступить к созданию его дочерних объектов. Когда все дочерние объекты созданы, и API-сервер вернул успешные ответы, контроллер может сохранить в поле status.phase значение Created. Затем, когда контроллер будет уведомлен об изменении ресурса AcmeNamespace, он сможет понять по этому полю, что ресурс уже был создан, и перейти к другим этапам согласования.

Такое использование поля status.phase для определения существующего состояния имеет один серьезный недостаток: подразумевается, что сам контроллер никогда не выйдет из строя. Но что, если при создании дочерних ресурсов возникнет проблема? Например, контроллер получит уведомление о новом экземпляре AcmeNamespace, создаст дочерний ресурс Namespace, но преждевременно завершит работу, не успев создать соответствующие ресурсы Role. Когда контроллер снова заработает, он обнаружит ресурс AcmeNamespace *без* значения Created в поле status.phase, попытается создать дочерний ресурс Namespace и получит ошибку, не имея возможности исправить положение. Чтобы этого не произошло, контроллер может записать в поле status.phase значение CreationInProgress непосредственно после обнаружения нового экземпляра AcmeNamespace. Таким образом, если в процессе создания возникнет сбой, то после возвращения к работе контроллер увидит стадию CreationInProgress и будет знать, что текущее состояние нельзя корректно определить по одному лишь разделу status. Для этого ему придется свериться с сопутствующими ресурсами кластера.

Когда текущее состояние нельзя установить на основе раздела status в AcmeNamespace, контроллер может обратиться к API-серверу (или лучше к его локальному кэшу объектов) за условиями, которые его интересуют. Обнаружив, что AcmeNamespace находится на стадии CreationInProgress, он может запросить у API-сервера сведения о существовании дочерних объектов, которые должны быть у этого ресурса. В представленном примере сбоя контроллер запросил бы информацию о дочернем экземпляре Namespace, убедился бы в его существовании и продолжил бы работу. Затем он повторил бы то же самое для ресурса Role, обнаружил бы, что тот *не* существует, и принялся бы за его создание. Таким образом мы можем сделать наш контроллер устойчивым к сбоям. Логику контроллера всегда следует создавать с расчетом на то, что сбой произойдет.

Более того, иногда наш контроллер может интересоваться текущим состоянием, выходящим за рамки кластера. Хороший пример — контроллеры облачной инфраструктуры. Состояние инфраструктурных систем нужно получать из API-интерфейса облачного провайдера за пределами кластера. Текущее состояние во многом зависит от назначения оператора, но обычно оно оказывается четким и понятным.

Желаемое состояние

Желаемое состояние системы выражено в разделах `spec` соответствующих ресурсов. Если взять наш оператор Namespace, то желаемое состояние, полученное из `namespaceName`, говорит контроллеру, каким должно быть поле `metadata.name` в итоговом экземпляре Namespace. Поле `adminUsername` определяет, каким должно быть значение `subjects[0].name` у `RoleBinding` `namespace-admin`. Это примеры прямой связи между желаемым состоянием и полями дочерних ресурсов. Но часто реализация оказывается не настолько очевидной.

Мы уже видели, как это происходит, в примере с `AcmeStore`, когда использовали поле `deploymentTier`. Это давало возможность пользователю указать одну-единственную переменную, которая информировала логику контроллера о том, какие значения нужно задать по умолчанию. Очень похожий принцип можно применить и к оператору Namespace. Наш новый, видоизмененный манифест `AcmeNamespace` показан в листинге 11.6.

Листинг 11.6. Манифест `AcmeNamespace` с новыми полями

```
apiVersion: tenancy.acme.com/v1alaphal
kind: AcmeNamespace
metadata:
  name: team-x
spec:
  namespaceName: app-y
  adminUsername: sam
  deploymentTier: dev ❶
```

❶ Новое поле в модели данных типа API AcmeNamespace.

Это заставит контроллер создать ресурс `ResourceQuota`, выглядящий так, как показано в листинге 11.7.

Листинг 11.7. Ресурс `ResourceQuota`, созданный для AcmeNamespace `team-x`

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev
spec:
  hard:
```

```
    cpu: "5"
    memory: 10Gi
    pods: "10"
```

Вместе с тем ресурс `ResourceQuota`, создаваемый по умолчанию для `deploymentTier: prod`, может выглядеть, как в листинге 11.8.

Листинг 11.8. Альтернативная версия `ResourceQuota`, которая создается, когда в AcmeNamespace указано `deploymentTier: prod`

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: prod
spec:
  hard:
    cpu: "500"
    memory: 200Gi
    pods: "100"
```

Согласование

В Kubernetes процесс согласования заключается в изменении текущего состояния с целью приведения его к желаемому. В простейшем случае это может выглядеть как запрос агента kubelet к среде выполнения для остановки контейнеров, относящихся к удаленному Pod'у. Бывают и более сложные ситуации, например, оператор может создать массив новых ресурсов в ответ на появление пользовательского ресурса, представляющего приложение, которое хранит свое состояние. Это примеры согласования, инициированного созданием или удалением ресурсов, которые описывают желаемое состояние. Но очень часто согласование происходит в ответ на изменение.

Примером простого изменения может служить увеличение числа реплик в ресурсе Deployment с 5 до 10. В текущем состоянии приложение имеет пять Pod'ов, а в желаемом состоянии их должно быть 10. Согласование, которое выполняет в этом случае контроллер Deployment, заключается в обновлении реплик в соответствующем объекте `ReplicaSet`. Затем контроллер `ReplicaSet` согласовывает состояние путем создания пяти новых ресурсов Pod, которые в свою очередь распределяются по узлам планировщиком, в результате чего подходящие агенты kubelet просят среду выполнения создать новые контейнеры.

Еще один, чуть более сложный пример — изменение образа в спецификации Deployment. Это обычно делается для обновления версии активного приложения. При согласовании состояния контроллер Deployment выполняет по умолчанию "плавающее" обновление. Он создает *новый* экземпляр `ReplicaSet` для новой версии приложения, инкрементирует в нем число реплик и декрементирует соответствующее число в старом экземпляре `ReplicaSet`, чтобы Pod'ы заменялись по одному. Когда все новые версии образа достигают желаемого числа реплик, процесс согласования завершается.

Процесс согласования для пользовательского контроллера, управляющего пользовательским ресурсом, в значительной степени зависит от типа этого ресурса. Неизменным остается только одно: если контроллеру не удается провести согласование ввиду обстоятельств, выходящих за рамки его предметной области, он должен повторять свои попытки, пока у него это не получится. В цикле согласования должно быть предусмотрено увеличение задержек между итерациями. Например, если вероятной причиной неудачного согласования является наличие в кластере других систем, которые сами активно согласовывают состояние, вы можете повторить попытку секундой позже. Но, чтобы избежать пустой траты ресурсов, рекомендуется экспоненциально увеличивать задержку с каждой итерацией, пока не будет достигнут какой-то разумный предел, скажем, 5 минут. Это позволит системам самостоятельно согласовать состояние и ограничит потребление ресурсов и сетевого трафика в случае, если им не удается сделать это быстро.

Аспекты реализации

В широком смысле для реализации базовых возможностей контроллера Namespace нам нужно:

- написать или сгенерировать манифест `AcmeNamespace`, как в предыдущем примере;
- отправить этот манифест API-серверу Kubernetes;
- сделать так, чтобы контроллер реагировал на создание ресурсов `Namespace`, `ResourceQuota`, `LimitRange`, `Roles` и `RoleBinding`.

Если использовать kubebuilder, то логика создания этих ресурсов будет находиться в методе `Reconcile`. Начальная процедура создания объекта `Namespace` с помощью контроллера может выглядеть так, как показано в листинге 11.9.

Листинг 11.9. Метод Reconcile для контроллера AcmeNamespace

```
// controllers/acmenamespace_controller.go

package controllers

import (
	"context"

	"github.com/go-logr/logr"
	corev1 "k8s.io/api/core/v1"
	metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
	"k8s.io/apimachinery/pkg/runtime"
	ctrl "sigs.k8s.io/controller-runtime"
	"sigs.k8s.io/controller-runtime/pkg/client"

	tenancyv1alpha1 "github.com/lander2k2/namespace-operator/api/v1alpha1"
)
...
```

```
func (r *AcmeNamespaceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error)
{
        ctx := context.Background()

        log := r.Log.WithValues("acmenamespace", req.NamespacedName)

        var acmeNs tenancyv1alpha1.AcmeNamespace ❶
        r.Get(ctx, req.NamespacedName, &acmeNs) ❷

        nsName := acmeNs.Spec.NamespaceName
        adminUsername := acmeNs.Spec.AdminUsername

        ns := &corev1.Namespace{ ❸
                ObjectMeta: metav1.ObjectMeta{
                        Name: nsName,
                        Labels: map[string]string{
                                "admin": adminUsername,
                        },
                },
        }

        if err := r.Create(ctx, ns); err != nil { ❹
                log.Error(err, "unable to create namespace")
                return ctrl.Result{}, err
        }

        return ctrl.Result{}, nil
}
...
```

❶ Переменная, которая будет представлять создаваемый, обновляемый и удаляемый объект `AcmeNamespace`.

❷ Извлечение содержимого объекта `AcmeNamespace` из запроса. Обработка ошибок опущена для краткости.

❸ Создание нового объекта `Namespace`.

❹ Создание нового ресурса `Namespace` в API-интерфейсе Kubernetes.

Этот упрощенный фрагмент кода демонстрирует, как контроллер создает новое пространство имен `namespace-admin`. Код для добавления объектов `Role` и `RoleBinding` в `namespace-admin` в общих чертах выглядит так, как показано в листинге 11.10.

Листинг 11.10. Создание `Role` и `RoleBinding` контроллером AcmeNamespace

```
// controllers/acmenamespace_controller.go
...
        role := &rbacv1.Role{
                ObjectMeta: metav1.ObjectMeta{
```

```
                        Name:      "namespace-admin",
                        Namespace: nsName,
                },
                Rules: []rbacv1.PolicyRule{
                        {
                                APIGroups: []string{"*"},
                                Resources: []string{"*"},
                                Verbs:     []string{"*"},
                        },
                },
        }

        if err := r.Create(ctx, role); err != nil {
                log.Error(err, "unable to create namespace-admin role")
                return ctrl.Result{}, err
        }

        binding := &rbacv1.RoleBinding{
                ObjectMeta: metav1.ObjectMeta{
                        Name:      "namespace-admin",
                        Namespace: nsName,
                },
                RoleRef: rbacv1.RoleRef{
                        APIGroup: "rbac.authorization.k8s.io",
                        Kind:     "Role",
                        Name:     "namespace-admin",
                },
                Subjects: []rbacv1.Subject{
                        {
                                Kind:      "User",
                                Name:      adminUsername,
                                Namespace: nsName,
                        },
                },
        }

        if err := r.Create(ctx, binding); err != nil {
                log.Error(err, "unable to create namespace-admin role binding")
                return ctrl.Result{}, err
        }

        return ctrl.Result{}, nil
}
...
```

На этом этапе мы уже готовы отправить манифест `AcmeNamespace` API-интерфейсу, и наш оператор создаст роль `Namespace` для `namespace-admin` и привяжет ее к предос-

тавленному нами имени пользователя с помощью RoleBinding. Как уже обсуждалось ранее, это хорошо работает при создании нового экземпляра AcmeNamespace, однако любые последующие попытки согласования (например, если объект AcmeNamespace изменится каким-либо образом) будут неудачными. При перезапуске контроллер должен заново составить список всех имеющихся ресурсов и согласовать их, если что-то поменялось. Поэтому на данном этапе простой перезапуск нарушит работу контроллера. Давайте это исправим, воспользовавшись новым полем в разделе status. Для начала в листинге 11.11 показан процесс добавления поля в AcmeNamespaceStatus.

Листинг 11.11. Добавление поля в состояние AcmeNamespace

```
// api/v1alpha1/acmenamespace_types.go

// AcmeNamespaceStatus определяет наблюдаемое состояние AcmeNamespace
type AcmeNamespaceStatus struct {

        // Отражает стадию, на которой находится AcmeNamespace
        // +optional
        // +kubebuilder:validation:Enum=CreationInProgress;Created
        Phase string `json:"phase"`
}

// +kubebuilder:object:root=true
// +kubebuilder:subresource:status
...
```

Теперь мы можем воспользоваться этим полем в нашем контроллере, как показано в листинге 11.12.

Листинг 11.12. Использование нового поля состояния в контроллере AcmeNamespace

```
// controllers/acmenamespace_controller.go
...

const (
        statusCreated    = "Created"
        statusInProgress = "CreationInProgress"
)

...

func (r *AcmeNamespaceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error)
{
        ...
```

```
        switch acmeNs.Status.Phase {
        case statusCreated:
                // ничего не делаем
                log.Info("AcmeNamespace child resources have been created")
        case statusInProgress:
                // TODO: запрашиваем и создаем по мере необходимости
                log.Info("AcmeNamespace child resource creation in progress")
        default:
                log.Info("AcmeNamespace child resources not created")

                // присваиваем состоянию значение statusInProgress
                acmeNs.Status.Phase = statusInProgress

                if err := r.Status().Update(ctx, &acmeNs); err != nil {
                        log.Error(err, "unable to update AcmeNamespace status")
                        return ctrl.Result{}, err
                }

                // создаем Namespace, Role и RoleBinding
                ...

                // присваиваем состоянию значение statusCreated
                acmeNs.Status.Phase = statusCreated
                if err := r.Status().Update(ctx, &acmeNs); err != nil {
                        log.Error(err, "unable to update AcmeNamespace status")
                        return ctrl.Result{}, err
                }
        }

        return ctrl.Result{}, nil
}

...
```

Теперь мы имеем контроллер, который может безопасно перезапускаться. Кроме того, у него уже есть зачатки системы для анализа текущего состояния с помощью дополнительного поля ресурса и учета этого состояния при согласовании.

Еще одно действие, которое обычно нужно выполнить, состоит в определении владельца дочерних ресурсов. Если сделать владельцем объектов `Namespace`, `Role` и `RoleBinding` ресурс `AcmeNamespace`, вместе с ним будут автоматически удаляться все его дочерние объекты. За этим будет следить API-сервер. Дочерние ресурсы будут удаляться даже в случае, если контроллер не запущен.

Это поднимает вопрос об области действия нашего типа API `AcmeNamespace`. При использовании Kubebuilder тип по умолчанию ограничивается пространством имен. Однако тип API, действующий на уровне пространства имен, не может владеть об-

щекластерным ресурсом таким, как Namespace. Kubebuilder позволяет использовать удобные маркеры, чтобы сгенерировать манифест CRD с подходящей областью действия для данного сценария использования (листинг 11.13).

Листинг 11.13. Обновленное определение API-интерфейса в проекте Kubebuilder

```
// api/v1alpha1/acmenamespace_types.go
package v1alpha1

import (
	metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// ОТРЕДАКТИРУЙТЕ ЭТОТ ФАЙЛ!  ЭТО ВАШ КАРКАС!
// ПРИМЕЧАНИЕ: теги json являются обязательными.  Их нужно указывать для любого
нового поля, подлежащего сериализации.

// AcmeNamespaceSpec определяет желаемое состояние AcmeNamespace
type AcmeNamespaceSpec struct {

	// Название пространства имен
	NamespaceName string `json:"namespaceName"`

	// Имя пользователя для namespace-admin
	AdminUsername string `json:"adminUsername"`
}

// AcmeNamespaceStatus определяет желаемое состояние AcmeNamespace
type AcmeNamespaceStatus struct {

	// Отражает стадию, на которой находится AcmeNamespace
	// +optional
	// +kubebuilder:validation:Enum=CreationInProgress;Created
	Phase string `json:"phase"`
}

// +kubebuilder:resource:scope=Cluster ❶
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

// AcmeNamespace описывает структуру API-интерфейса acmenamespaces
type AcmeNamespace struct {
	metav1.TypeMeta   `json:",inline"`
	metav1.ObjectMeta `json:"metadata,omitempty"`

	Spec AcmeNamespaceSpec     `json:"spec,omitempty"`
	Status AcmeNamespaceStatus `json:"status,omitempty"`
}
```

```
// +kubebuilder:object:root=true

// AcmeNamespaceList содержит список объектов AcmeNamespace
type AcmeNamespaceList struct {
        metav1.TypeMeta `json:",inline"`
        metav1.ListMeta `json:"metadata,omitempty"`
        Items           []AcmeNamespace `json:"items"`
}

func init() {
        SchemeBuilder.Register(&AcmeNamespace{}, &AcmeNamespaceList{})
```

❶ Этот маркер выберет подходящую область действия для CRD при генерации манифеста с помощью команды `make manifests`.

В результате будет сгенерирован CRD, похожий на тот, что показан в листинге 11.14.

Листинг 11.14. Общекластерный CRD для типа API AcmeNamespace

```
---
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: (devel)
  creationTimestamp: null
  name: acmenamespaces.tenancy.acme.com
spec:
  group: tenancy.acme.com
  names:
    kind: AcmeNamespace
    listKind: AcmeNamespaceList
    plural: acmenamespaces
    singular: acmenamespace
  scope: Cluster ❶
  subresources:
    status: {}
  validation:
    openAPIV3Schema:
      description: AcmeNamespace is the Schema for the acmenamespaces API
      properties:
        apiVersion:
          description: 'APIVersion defines the versioned schema of this
            representation of an object. Servers should convert recognized
            schemas to the latest internal value, and may reject unrecognized
            values.'
          type: string
```

```
        kind:
          description: 'Kind is a string value representing the REST resource this
            object represents. Servers may infer this from the endpoint the client
            submits requests to. Cannot be updated. In CamelCase.'
          type: string
        metadata:
          type: object
        spec:
          description: AcmeNamespaceSpec defines the desired state of
AcmeNamespace
          properties:
            adminUsername:
              description: The username for the namespace admin
              type: string
            namespaceName:
              description: The name of the namespace
              type: string
          required:
          - adminUsername
          - namespaceName
          type: object
        status:
          description: 'AcmeNamespaceStatus defines the observed state of
                        AcmeNamespace'
          properties:
            phase:
              description: Tracks the phase of the AcmeNamespace
              enum:
              - CreationInProgress
              - Created
              type: string
          type: object
      type: object
  version: v1alpha1
  versions:
  - name: v1alpha1
    served: true
    storage: true
status:
  acceptedNames:
    kind: ""
    plural: ""
  conditions: []
  storedVersions: []
```

❶ Выбор подходящей области действия для ресурса.

Теперь мы можем сделать ресурс AcmeNamespace владельцем всех дочерних ресурсов. В результате в разделе metadata каждого дочернего ресурса появится поле ownerReferences. На этом этапе наш метод Reconcile выглядит так, как в листинге 11.15.

Листинг 11.15. Определение параметров владения для дочерних ресурсов объекта AcmeNamespace

```
func (r *AcmeNamespaceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error)
{
        ctx := context.Background()
        log := r.Log.WithValues("acmenamespace", req.NamespacedName)

        var acmeNs tenancyv1alpha1.AcmeNamespace
        if err := r.Get(ctx, req.NamespacedName, &acmeNs); err != nil {
                if apierrs.IsNotFound(err) { ❶
                        log.Info("resource deleted")
                        return ctrl.Result{}, nil
                } else {
                        return ctrl.Result{}, err
                }
        }

        nsName := acmeNs.Spec.NamespaceName
        adminUsername := acmeNs.Spec.AdminUsername

        switch acmeNs.Status.Phase {
        case statusCreated:
                // ничего не делаем
                log.Info("AcmeNamespace child resources have been created")
        case statusInProgress:
                // TODO: запрашиваем и создаем по мере необходимости
                log.Info("AcmeNamespace child resource creation in progress")
        default:
                log.Info("AcmeNamespace child resources not created")

                // присваиваем состоянию значение statusInProgress
                acmeNs.Status.Phase = statusInProgress
                if err := r.Status().Update(ctx, &acmeNs); err != nil {
                        log.Error(err, "unable to update AcmeNamespace status")
                        return ctrl.Result{}, err
                }

                ns := &corev1.Namespace{
                        ObjectMeta: metav1.ObjectMeta{
                                Name: nsName,
                                Labels: map[string]string{
                                        "admin": adminUsername,
```

```
                },
            },
        }

        // указываем ссылку на владельца для пространства имен ❷
        err := ctrl.SetControllerReference(&acmeNs, ns, r.Scheme)
    if err != nil {
            log.Error(err, "unable to set owner reference on
namespace")
            return ctrl.Result{}, err
        }

        if err := r.Create(ctx, ns); err != nil {
            log.Error(err, "unable to create namespace")
            return ctrl.Result{}, err
        }

        role := &rbacv1.Role{
            ObjectMeta: metav1.ObjectMeta{
                Name:      "namespace-admin",
                Namespace: nsName,
            },
            Rules: []rbacv1.PolicyRule{
                {
                    APIGroups: []string{"*"},
                    Resources: []string{"*"},
                    Verbs:     []string{"*"},
                },
            },
        }

        // указываем ссылку на владельца для роли ❸
        err = ctrl.SetControllerReference(&acmeNs, role, r.Scheme)
    if err != nil {
            log.Error(err, "unable to set owner reference on role")
            return ctrl.Result{}, err
        }

        if err := r.Create(ctx, role); err != nil {
            log.Error(err, "unable to create namespace-admin role")
            return ctrl.Result{}, err
        }

        binding := &rbacv1.RoleBinding{
            ObjectMeta: metav1.ObjectMeta{
                Name:      "namespace-admin",
```

```
                Namespace: nsName,
            },
            RoleRef: rbacv1.RoleRef{
                APIGroup: "rbac.authorization.k8s.io",
                Kind:     "Role",
                Name:     "namespace-admin",
            },
            Subjects: []rbacv1.Subject{
                {
                    Kind:      "User",
                    Name:      adminUsername,
                    Namespace: nsName,
                },
            },
        }

        // указываем ссылку на владельца для RoleBinding ❹
        err = ctrl.SetControllerReference(&acmeNs, binding, r.Scheme);
    if err != nil {
            log.Error(err, "unable to set reference on role binding")
            return ctrl.Result{}, err
        }

        if err := r.Create(ctx, binding); err != nil {
            log.Error(err, "unable to create role binding")
            return ctrl.Result{}, err
        }

        // присваиваем состоянию значение statusCreated
        acmeNs.Status.Phase = statusCreated
        if err := r.Status().Update(ctx, &acmeNs); err != nil {
            log.Error(err, "unable to update AcmeNamespace status")
            return ctrl.Result{}, err
        }
    }

    return ctrl.Result{}, nil
}
...
```

❶ Проверяем, был ли найден ресурс. Если нет, не пытаемся выполнить согласование при удалении AcmeNamespace.

❷ Указываем ссылку на владельца для Namespace.

❸ Указываем ссылку на владельца для Role.

❹ Указываем ссылку на владельца для RoleBinding.

Обратите внимание на то, что мы добавили проверку ошибок, чтобы узнать, был ли найден ресурс `AcmeNamespace`. Это сделано в связи с тем, что при удалении данного ресурса обычный процесс согласования завершится неудачей, так как у нас больше не будет желаемого состояния. В этом случае мы указываем ссылку на владельца в дочернем ресурсе, чтобы API-сервер позаботился о согласовании состояния для событий удаления.

Это иллюстрация того, что в процессе согласования нельзя делать предположения о текущем состоянии. Согласование инициируется, когда:

- запускается или перезапускается контроллер;
- создается ресурс;
- вносится изменение в ресурс, в том числе и самим контроллером;
- удаляется ресурс;
- выполняется периодическая синхронизация с API-интерфейсом, чтобы обеспечить корректное представление системы.

В связи с этим позаботьтесь о том, чтобы ваш процесс согласования не был основан на предположениях о событии, которое его инициировало. Используйте поле `status`, определяйте существенные условия в других ресурсах, если это необходимо, и согласовывайте состояние соответствующим образом.

Веб-хуки допуска

Если вашему ресурсу нужны параметры по умолчанию или механизм проверки, который нельзя реализовать с помощью спецификации OpenAPI v3 в CRD, который создает ваш новый тип API, можете воспользоваться проверяющими и изменяющими веб-хуками. У утилиты командной строки Kubernetes есть команда `create webhook`, предназначенная специально для таких случаев. Она генерирует шаблонный код, чтобы вы могли быстрее создать свой веб-хук.

Чтобы понять, чем проверяющий веб-хук может быть полезен для нашего примера с оператором `Namespace` и его ресурсом `AcmeNamespace`, можно взять проверку поля `adminUsername`. Для удобства ваш веб-хук может обращаться к вашему корпоративному провайдеру идентификации, чтобы гарантировать корректность предоставленного имени пользователя и предотвратить ошибки, для исправления которых требуется человеческое вмешательство.

В качестве примера параметра по умолчанию можно взять наиболее распространенное значение для `deploymentTier` – `dev`. Это особенно полезно для поддержания обратной совместимости с определениями существующих ресурсов, когда вносится изменение, которое добавляет новые поля в модель данных пользовательского ресурса.

Веб-хуки допуска редко встречаются в прототипах или выпусках операторов, предшествующих alpha-версии, но их часто задействуют для улучшения удобства взаимодействия в стабильных выпусках проекта. Тема управления допуском подробно обсуждается в *главе 8*.

Финализаторы

Мы рассмотрели примеры того, как сделать пользовательский ресурс владельцем дочернего ресурса, чтобы второй всегда удалялся вместе с первым. Однако такой механизм не всегда эффективен. Если у пользовательского ресурса есть связи с другими компонентами кластера, и понятие владения для этих связей не подходит, или если при удалении вашего пользовательского ресурса необходимо обновить условия за пределами кластера, вам, скорее всего, нужно использовать финализаторы.

Финализаторы указываются в разделе `finalizers` метаданных ресурса, как показано в листинге 11.16.

Листинг 11.16. Манифест AcmeNamespace с финализатором

```
apiVersion: tenancy.acme.com/v1alapha1
kind: AcmeNamespace
metadata:
  name: team-x
  finalizers:
  - namespace.finalizer.tenancy.acme.com ❶
spec:
  namespaceName: app-y
  adminUsername: sam
```

❶ Строковое значение, используемое в качестве финализатора.

Строковое значение, указанное в качестве финализатора, не касается никаких других компонентов системы, кроме вашего контроллера. Просто выберите строку, которая точно будет уникальной, на случай, если другим контроллерам понадобится применить финализаторы к тому же ресурсу.

Если у ресурса есть какой-либо финализатор, API-сервер не будет его удалять. При получении запроса на удаление он просто обновит этот ресурс, добавив в его метаданные поле `deletionTimestamp`. Это инициирует процесс согласования в вашем контроллере. В метод `Reconcile` контроллера нужно добавить проверку поля `deletionTimestamp`, чтобы он мог завершить все операции, которые нужно выполнить перед удалением. Когда операции завершены, контроллер может убрать финализатор. Это проинформирует API-сервер о том, что ресурс уже можно удалить.

Операции, предшествующие удалению, зачастую выполняются в системах за пределами кластера. Вернемся к примеру с оператором `Namespace`. Если у вас есть корпоративная система учета финансовых расходов, которая следит за использованием ресурса `Namespace` и должна обновляться при его удалении, финализатор может сделать так, чтобы ваш оператор обновил внешнюю систему, прежде чем удалять `Namespace`. Еще один распространенный пример — ситуация, когда приложение использует в рамках приложения управляемые сервисы, такие как базы данных или объектные хранилища. Когда удаляется экземпляр приложения, ресурсы этих управляемых сервисов тоже, скорее всего, должны быть освобождены.

Расширение планировщика

Планировщик отвечает за ключевые функции Kubernetes. Ценность Kubernetes во многом состоит в абстрагировании пулов серверов для выполнения приложений. Именно планировщик принимает решение о том, где будут выполняться Pod'ы. Будет справедливо сказать, что этот контроллер вместе с kubelet является "сердцем" Kubernetes, вокруг которого построено все остальное. Планировщик служит основополагающим сервисом вашей платформы приложений. В этом разделе вы увидите, как его модифицировать, расширять и изменять его поведение.

Будет полезно провести параллели между основными компонентами плоскости управления, такими как планировщик, и пользовательскими операторами, которые мы рассматривали ранее в этой главе. В обоих случаях мы имеем дело с контроллерами Kubernetes, которые управляют ресурсами кластера. Если говорить о пользовательских операторах, то мы создаем совершенно новые пользовательские контроллеры, в то время как планировщик входит в состав ядра Kubernetes и развертывается вместе с каждым кластером. Для пользовательских операторов разрабатываются и создаются новые пользовательские ресурсы, тогда как планировщик управляет встроенным ресурсом Pod.

Наш опыт показывает, что у пользователей Kubernetes редко возникает необходимость в расширении планировщика или изменении его поведения. Но, учитывая, насколько важную роль в работе кластера играет этот компонент, будет целесообразно обсудить то, как планировщик принимает свои решения, и как их видоизменить в случае необходимости. Стоит еще раз подчеркнуть, что гениальность проекта Kubernetes во многом объясняется его расширяемостью и модульностью. Если планировщик не отвечает вашим потребностям, то вы можете его модифицировать, дополнить его поведение и вовсе его заменить.

Исследуя эту тему, мы поговорим о том, как планировщик определяет, где размещать Pod'ы. Это поможет нам понять, как принимается каждое решение планировщика, и каким образом мы можем влиять на эти решения с помощью политик планирования. Мы также обсудим вариант с использованием нескольких планировщиков и даже напишем наш собственный контроллер планирования.

Предикаты и приоритеты

Прежде чем переходить к расширению или изменению планировщика, необходимо понимать, как он принимает свои решения. Для определения того, на каком Узле разметить Pod, планировщик использует двухэтапный процесс.

На первом этапе проводится фильтрация. Планировщик отбрасывает Узлы, которые не годятся для размещения Pod'а с помощью ряда предикатов. Например, один предикат проверяет, допускает ли развертываемый Pod ограничения Узла. Узлы плоскости управления обычно имеют ограничение, которое не дает развертывать на них обычные приложения. Если Pod не допускает никаких ограничений, все потенциальные Узлы для развертывания будут отфильтрованы. Еще один предикат проверяет, достаточно ли у Узла ресурсов процессора и памяти для любого Pod'а, ко-

торый запрашивает определенное их количество. Если у Узла не хватает ресурсов, чтобы удовлетворить требованиям спецификации Pod'а, он отклоняется. Если подходящих Узлов не нашлось, Pod остается в состоянии `Pending` до изменения обстоятельств, например, пока в кластер не будет добавлен новый подходящий Узел. Если найден один Узел, развертывание может начаться сразу. Если подходящих Узлов несколько, планировщик переходит ко второму этапу.

Второй этап заключается в определении *самого* подходящего Узла для заданного Pod'а. Для этого используются приоритеты. Одним из приоритетов, которые делают Узел более подходящим, является наличие образа контейнера, используемого Pod'ом. Еще один такой приоритет состоит в отсутствии других Pod'ов, входящих в состав того же Сервиса. Это означает, что планировщик пытается распределить Pod'ы одного Сервиса по разным Узлам, чтобы повысить их отказоустойчивость. Кроме того, на этом этапе реализуются правила подобия `preferred*`. По его завершении каждому подходящему Узлу назначается балл. Узел с самым высоким баллом считается лучшим выбором для Pod'а и служит для его развертывания.

Политики планирования

Политики планирования предназначены для конфигурации предикатов и приоритетов, которыми будет руководствоваться планировщик. Конфигурационный файл с политикой планирования внутри можно записать на файловую систему Узлов плоскости управления и предоставить планировщику флаг `--policy-config-file`, но вместо этого лучше воспользоваться ресурсом `ConfigMap`. Передайте планировщику флаг `--policy-configmap`, и впоследствии вы сможете обновлять политику планирования посредством API-сервера. Отметим, что в случае выбора второго метода вам, скорее всего, нужно будет добавить в `ClusterRole system:kube-scheduler` правило для получения объектов `ConfigMap`.



На момент написания этих строк оба флага планировщика, `--policy-config-file` и `--policy-configmap`, по-прежнему работают, но в официальной документации они помечены как устаревшие. В связи с этим, если вы хотите изменить поведение планировщика, мы рекомендуем использовать не политики, описанные здесь, а профили планирования, о которых пойдет речь в следующем разделе.

Например, политика `ConfigMap`, показанная в листинге 11.17, позволит Pod'у выбрать Узел с помощью поля `nodeSelector`, только в том случае, если у этого Узла есть метка с ключом `selectable`.

Листинг 11.17. Пример ресурса `ConfigMap`, определяющего политику планирования

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: scheduler-policy-config
  namespace: kube-system
```

```
data:
  policy.cfg: |+ ❶
    apiVersion: v1
    kind: Policy
    predicates:
    - name: "PodMatchNodeSelector" ❷
      argument:
        labelsPresence:
          labels:
          - "selectable" ❸
          presence: true ❹
```

❶ Имя файла, в котором планировщик будет искать политики.

❷ Предсказуемое имя для селекторов `nodeSelector`.

❸ Ключ метки, с помощью которой вы хотите добавить ограничение в процесс выбора. В данном примере узел, у которого нет этого ключа метки, не сможет быть выбран Pod'ом.

❹ Это говорит о том, что указанная метка должна присутствовать. Если вы хотите, чтобы она отсутствовала, нужно указать `false`. Если задать эту конфигурацию с полем `presence: true`, то Узел без метки `selectable: ""` не сможет быть выбран Pod'ом.

При реализации этой политики планирования Pod, описанная в манифесте из листинга 11.18, будет развернута на подходящем Узле только в случае, если у того есть сразу две метки: `device: gpu` и `selectable: ""`.

Листинг 11.18. Манифест Pod'а с полем `nodeSelector` для управления процессом планирования

```
apiVersion: v1
kind: Pod
metadata:
  name: terminator
spec:
  containers:
  - image: registry.acme.com/skynet/t1000:v1
    name: terminator
  nodeSelector:
    device: gpu
```

Профили планирования

Профили планирования позволяют включать и выключать модули, встроенные в планировщик. Чтобы задать профиль, при запуске планировщика можно указать имя файла с помощью флага `--config`. Эти модули реализуют различные механизмы расширения, в число которых среди прочих входят этапы фильтрации и расстановки приоритетов, рассмотренные ранее. Как показывает наш опыт, необходимость в конфигурации планировщика таким способом возникает редко. Но, если вам нужно будет это сделать, инструкции можно найти в документации Kubernetes.

Несколько планировщиков

Следует отметить, что вы можете не ограничиваться одним планировщиком. Вы можете развернуть любое число планировщиков, либо созданных вами, либо встроенных в Kubernetes и имеющих разные политики и профили. В спецификации Pod'a можно указать поле `schedulerName`, которое определяет, какой компонент будет заниматься ее планированием. Учитывая повышенную сложность этой модели, для приложений с такими необычными требованиями к планированию лучше назначить отдельные кластеры.

Создание собственного планировщика

В тех редких случаях, когда вам не подходит планировщик Kubernetes, даже с применением политик и профилей, у вас остается возможность разработать свой собственный компонент для планирования. Для этого вам нужно будет создать контроллер, который следит за ресурсами Pod и в момент их создания определяет, где они должны быть развернуты, обновляя при этом их поле `nodeName`. Несмотря на узкие рамки, это задача не из простых. Как мы уже убедились, встроенный планировщик представляет собой развитой контроллер, который постоянно учитывает множество сложных факторов, принимая решения о планировании. Если ваши требования достаточно специфичные для того, чтобы оправдать создание собственного планировщика, то вам, скорее всего, придется потратить существенные инженерные ресурсы на совершенствование его поведения. Советуем использовать этот подход только в случае, если вам не подходит ни один из существующих планировщиков, и вы можете привлечь к этому проекту профессионалов с глубокими знаниями Kubernetes.

Резюме

Необходимо понимать, какие механизмы расширения доступны в Kubernetes, и как лучше всего добавлять в платформу новые сервисы, необходимые для удовлетворения требований ваших пользователей. Изучите шаблон проектирования "оператор" и то, как он применяется в Kubernetes. Если у вас есть реальная необходимость в создании оператора, определитесь с тем, какие средства разработки и язык вы будете использовать, спроектируйте модель данных своего пользовательского ресурса и затем напишите контроллер, который будет этим ресурсом управлять. Наконец, если стандартное поведение планировщика вас не удовлетворяет, попробуйте его изменить с помощью политик и профилей планирования. В крайнем случае у вас остается возможность разработать собственный планировщик, который заменит встроенный или будет работать параллельно с ним.

Реализуя принципы и методы, описанные в этой главе, вы сможете больше не ограничиваться вспомогательными инструментами и программным обеспечением, предоставляемые сообществом или поставщиками вашей компании. Если у вас возникнут важные задачи, для которых нет готовых решений, в вашем распоряжении есть инструменты и инструкции для добавления любых специализированных сервисов платформы, которые могут понадобиться вашей организации.

ГЛАВА 12

Мультитенантность

При создании платформы приложений поверх Kubernetes необходимо продумать работу с клиентами, которые будут ею пользоваться. Как уже неоднократно упоминалось в этой книге, Kubernetes предоставляет набор основополагающих возможностей, с помощью которых можно удовлетворить множество требований, в том числе и совместное размещение рабочих заданий. Kubernetes предлагает различные механизмы, с помощью которых вы можете обеспечить безопасное сосуществование приложений на одной платформе. Вместе с тем в Kubernetes не существует такого понятия, как "арендатор" (или "клиент"). Это может быть приложение, команда разработчиков, бизнес-подразделение или что-то другое. Конкретный смысл, который вкладывается в этот термин, должны определить вы и ваша организация. Надеемся, данная глава вам в этом поможет.

Определившись с тем, кто ваши клиенты, вы должны решить, должны ли они размещаться на одной и той же платформе. Помогая крупным организациям с созданием платформ приложений, мы обнаружили, что те, кто работает над платформой, обычно заинтересованы в том, чтобы сделать ее мультитенантной. В то же время данное решение сильно зависит от того, с какими пользователями вы имеете дело, и какой уровень доверия между ними существует. Например, предприятие, использующее общую платформу приложений — это одно, а компания, предлагающая внешним клиентам контейнеры как услугу — совсем другое.

В этой главе мы сначала исследуем, насколько изолированными можно сделать пользователей в Kubernetes. Характеристики ваших приложений и конкретные требования будут диктовать степень изоляции, которую вам нужно будет обеспечить. Чем сильнее изоляция, тем больше вложений вам придется сделать в этой области. Вслед за этим мы обсудим пространства имен, основополагающий составной элемент, на который приходится значительная часть функций мультитенантности Kubernetes. В завершение мы подробно рассмотрим разные возможности Kubernetes, с помощью которых можно изолировать клиентов в кластере, включая управление доступом на основе ролей (англ. Role-Based Access Control или RBAC), запросы и лимиты на ресурсы, политики безопасности Pod'ов и др.

Уровни изоляции

Платформа Kubernetes совместима с разными моделями использования, каждая из которых имеет свои плюсы и минусы. Самый важный фактор при выборе модели — степень изоляции, которая требуется вашим приложениям. Например, выполнение непроверенного кода, написанного разными сторонними разработчиками,

требует более строгой изоляции в сравнении с размещением внутренних приложений вашей собственной организации. В целом можно выделить две основные модели использования: однотенантные и мультитенантные кластеры. Давайте обсудим сильные и слабые стороны каждой из них.

Однотенантные кластеры

Однотенантная модель (рис. 12.1) обеспечивает самую строгую изоляцию между клиентами, так как ресурсы кластера не используются совместно. Этот подход заманчив тем, что вам не нужно решать сложные проблемы, связанные с мультитенантностью. Иными словами, проблемы с изоляцией отсутствуют в принципе.

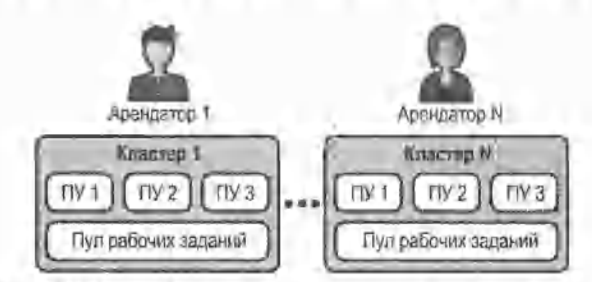


Рис. 12.1. Каждый клиент выполняется в отдельном кластере (ПУ — это плоскость управления узла)

Однотенантные кластеры годятся для систем с небольшим числом клиентов. Но данной модели присущи следующие потенциальные недостатки:

- *Накладные расходы.* У каждого однотенантного кластера должна быть своя собственная плоскость управления, которой в большинстве случаев требуется по меньшей мере три отдельных узла. Чем больше у вас клиентов, тем больше ресурсов будет выделяться плоскостям управления — ресурсов, которые можно было бы направить на выполнение приложений. Кроме того, в каждом кластере есть набор приложений, которые предоставляют сервисы платформы. Эти сервисы влекут за собой расход дополнительных ресурсов, так как их можно было бы сделать общими для разных клиентов в мультитенантном кластере. Хорошие примеры — средства мониторинга, контроллеры политик (такие как Open Policy Agent) и контроллеры Ingress.
- *Повышенная сложность администрирования.* Управление большим числом кластеров может стать непростой задачей для администраторов платформы. Каждый кластер нужно развертывать, отслеживать, обновлять и т. д. Представьте, насколько сложно было бы исправить уязвимость безопасности в сотнях кластеров. Чтобы делать это эффективно, необходимы вложения в дополнительный инструментарий.

Но, даже несмотря на перечисленные недостатки, мы видели множество успешных реализаций однотенантных кластеров. И по мере того как становятся зрелыми такие инструменты управления жизненным циклом кластера, как Cluster API (**https://oreil.ly/8QRz7**), однотенантная модель стала более доступной для внедре-

ния. Тем не менее, помогая организациям, мы в основном имеем дело с мультитенантными кластерами, которые обсуждаются далее.

Мультитенантные кластеры

Недостатки рассмотренной однотенантной модели могут восполнить кластеры с несколькими клиентами. Вместо того чтобы развертывать и администрировать отдельный кластер для каждого клиента, администраторы платформы могут сосредоточиться на небольшом числе кластеров, что снижает накладные расходы и упрощает администрирование (рис. 12.2). Но у этого есть и обратная сторона. Реализация мультитенантных кластеров более сложная и тонкая, так как вам нужно позаботиться о том, чтобы клиенты могли сосуществовать, не мешая друг другу.



Рис. 12.2. Один кластер, разделяемый несколькими клиентами (ПУ — это плоскость управления узла)

Мультитенантность можно разделить на две общие категории: строгая и нестрогая. *Нестрогая мультитенантность*, которую иногда называют "мультикомандностью", предполагает, что между клиентами платформы существует некая степень доверия. Подобная модель уместна, когда клиенты относятся к одной организации. Например, реальная платформа приложений с разными клиентами обычно может применять принципы нестрогой мультитенантности. Это связано с тем, что клиенты заинтересованы в хороших взаимоотношениях, так как успех организации является их общей целью. Но, несмотря на хорошие побуждения, изоляция клиентов все равно необходима, учитывая возможность возникновения проблем непроизвольного характера (таких как уязвимости, программные ошибки и т. д.).

С другой стороны, модель *строгой мультитенантности* основана на отсутствии доверия между клиентами. Более того, с точки зрения безопасности, клиенты считаются противниками, чтобы гарантировать использование адекватных механизмов изоляции. Хороший пример — платформа, которая выполняет непроверенный код, принадлежащий разным организациям. В таких случаях строгая изоляция между клиентами носит обязательный характер, позволяя им безопасно сосуществовать в одном кластере.

Если вернуться к нашей аналогии с жильем, которую мы упоминали в *главе 1*, можно сказать, что модель *нестрогой мультитенантности* — это эквивалент семейства, проживающего в частном доме. Члены семьи пользуются общей кухней, гостиной и коммунальными услугами, но у каждого из них есть своя спальня. Для

сравнения, модель строгой мультитенантности больше напоминает многоквартирный дом с разными семьями, каждая из которых живет за входной дверью с замком.

Понятия строгости и нестрогости могут быть полезными при обсуждении мультитенантности, однако на практике все не так очевидно. К мультиатенантности лучше всего относиться как к области, на одном конце которой изоляции нет совсем, и клиенты платформы не ограничены в своих действиях и могут потреблять все ее ресурсы, а на другом конце мы имеем полную изоляцию, когда клиенты жестко контролируются и изолируются на всех уровнях платформы.

Как можно себе представить, создание мультитенантной платформы без изоляции между клиентами нецелесообразно. С другой стороны, разработка мультитенантной платформы с полной изоляцией клиентов может оказаться дорогостоящей (или даже тщетной). В связи с этим необходимо найти тот участок спектра мультитенантности, который подходит для ваших приложений и вашей организации в целом.

Чтобы определить, какая степень изоляции нужна вашим приложениям, необходимо учитывать разные уровни платформы на основе Kubernetes, которые *могут* быть изолированы:

- *Плоскость рабочих заданий* — эта плоскость состоит из узлов, на которых могут выполняться рабочие задания. В мультитенантной системе приложения обычно распределяются по общему пулу узлов. Изоляция на этом уровне подразумевает равномерное распределение ресурсов, механизмов безопасности, границ сети и т.д.
- *Плоскость управления* — охватывает компоненты, из которых состоит кластер Kubernetes, включая API-сервер, диспетчер контроллеров и планировщик. В Kubernetes предусмотрены разные механизмы для разделения клиентов на этом уровне, такие как авторизация (т. е. RBAC), управление доступом, а также приоритеты API-интерфейса и принцип равнодоступности.
- *Сервисы платформы* — это централизованные средства журналирования, мониторинга, управления входящим трафиком, внутрикластерный DNS-сервер и т.д. Они тоже могут потребовать некоторой степени изоляции, в зависимости от ваших приложений. Например, вы можете запретить клиентам анализировать журналы друг друга или обнаруживать чужие сервисы с помощью кластерного DNS-сервера.

Kubernetes предоставляет разные компоненты, с помощью которых можно обеспечить изоляцию на каждом из этих уровней. Но, прежде чем в них углубляться, обсудим пространства имен — основополагающее средство разделения клиентов в кластере.

Разделение на основе пространств имен

Пространства имен делают возможным целый ряд функций API-интерфейса Kubernetes. Они позволяют организовать ваш кластер, обеспечить выполнение по-

литики, управлять доступом и т. д. Что более важно, они являются важнейшими составными элементами реализации мультитенантной платформы Kubernetes, лежащими в основе размещения и изоляции клиентов.

Но, когда речь идет об изоляции клиентов, необходимо помнить о том, что пространство имен — это логическая концепция плоскости управления Kubernetes. Само по себе, без политики или конфигурации, оно никак не влияет на рабочую плоскость. Например, в отсутствие дополнительных ограничений планирования приложения, принадлежащие разным пространствам имен, могут с большой долей вероятности выполняться на одном и том же узле. В конце концов, пространство имен — это всего лишь элемент метаданных, который назначается ресурсам в API-интерфейсе Kubernetes.

С другой стороны, многие механизмы изоляции, которые мы исследуем в этой главе, завязаны на понятии пространства имен. Примерами таких механизмов являются RBAC, квоты на ресурсы и сетевые политики. В связи с этим одно из первых решений, которое необходимо принять в ходе разработки стратегии использования, состоит в определении принципов использования пространства имен. Помогая реальным организациям, мы сталкиваемся со следующими подходами:

- *Отдельное пространство имен для каждой команды.* В рамках такой модели каждая команда имеет доступ только к одному пространству имен. Это позволяет легко применять политики и квоты к отдельным командам. Но, если команда владеет множеством сервисов, ей может быть непросто работать внутри всего одного пространства. В целом наш опыт показывает, что данная модель подходит для небольших организаций, которые начинают внедрять Kubernetes.
- *Отдельное пространство имен для каждого приложения.* Согласно этому подходу, каждое приложение в кластере получает отдельное пространство имен, что упрощает применение политик и квот к определенным приложениям. Недостаток подобной модели состоит в том, что клиенты обычно получают доступ сразу к нескольким пространствам имен, что может усложнить их добавление в кластер и применение к ним политик и квот. Тем не менее, это, наверное, наиболее подходящий вариант для организаций и предприятий, которые разрабатывают мультитенантные платформы.
- *Отдельное пространство имен для каждого уровня.* Эта модель делит среды выполнения (или окружения) на уровни с помощью пространств имен. Мы обычно отказываемся от такого подхода, предпочитая использовать для уровней разработки, финального тестирования и эксплуатации отдельные кластеры.

Выбор того или иного варианта во многом зависит от ваших требований к изоляции и от структуры вашей компании. Если вы склоняетесь к модели с отдельным пространством имен для каждой команды, помните о том, что каждый член команды и каждое приложение будет иметь доступ ко всем ресурсам в этом пространстве. Например, если Элис и Боб работают в одной команде и авторизованы для получения объектов `Secret` в пространстве имен этой команды, то ничто не помешает Элис просматривать объекты Боба.

Мультитенантность в Kubernetes

До сих пор мы обсуждали разные модели использования, которые можно реализовать при создании платформы на основе Kubernetes. Оставшаяся часть этой главы посвящена мультитенантным кластерам и различным возможностям Kubernetes, которые позволяют безопасно и эффективно размещать клиентов. На страницах следующих разделов вы заметите, что некоторые из описываемых возможностей уже были рассмотрены в других главах. В таких случаях мы пройдемся по ним еще раз, делая основной упор на их отношение к мультитенантности.

Вначале мы сосредоточимся на механизмах изоляции, доступных на уровне плоскости управления. Речь в основном об RBAC, квотах на ресурсы и проверяющих веб-хуках допуска. Затем мы перейдем к плоскости рабочих заданий и обсудим запросы и ограничения ресурсов, сетевые политики и политики безопасности Pod'ов. В конце в качестве примера сервисов платформы, которые можно спроектировать с учетом мультитенантности, будут рассмотрены мониторинг и централизованное журналирование.

Управление доступом на основе ролей

Если вы размещаете несколько клиентов в одном и том же кластере, вам нужно обеспечить изоляцию на уровне API-сервера, чтобы клиенты не могли изменять ресурсы, которые им не принадлежат. Механизм авторизации RBAC (Role-Based Access Control — управление доступом на основе ролей) позволяет сконфигурировать такую политику. Как уже обсуждалось в *главе 10*, API-сервер поддерживает разные механизмы идентификации пользователей. После подтверждения учетные данные клиента передаются системе RBAC, которая определяет, авторизован ли он для выполнения запрошенного действия.

Размещая клиентов в своем кластере, вы можете выдавать им права доступа к одному или нескольким пространствам имен, в которых они могут создавать и администрировать ресурсы API-интерфейса. Для авторизации каждого клиента его учетные данные должны быть привязаны к объектам `Role` или `ClusterRole`. Это достигается с помощью ресурса `RoleBinding`. В листинге 12.1 приведен пример такого ресурса, который предоставляет Группе `app1-viewer` доступ на чтение к пространству имен `app1`. Избегайте использования объектов `ClusterRoleBinding` для клиентов (если только он не подходит для вашего конкретного случая), так как он позволяет клиентам применять привязанные к ним роли во всех пространствах имен.

Листинг 12.1

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: viewers
  namespace: app1
```

```
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: view
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: app1-viewer
```

Вы можете заметить, что ресурс `RoleBinding` ссылается на объект `ClusterRole` с именем `view`. Это встроенная роль, доступная в Kubernetes. Kubernetes предлагает ряд встроенных ролей для распространенных сценариев:

- `view` — дает клиентам доступ на чтение ресурсов внутри пространства имен. Эту роль, к примеру, можно привязать ко всем разработчикам в команде, что позволит им анализировать и диагностировать свои ресурсы в кластерах.
- `edit` — позволяет клиентам не только читать, но также создавать, изменять и удалять ресурсы внутри пространства имен. Учитывая возможности этой роли, ее привязка во многом зависит от вашего подхода к развертыванию приложений.
- `admin` — помимо просмотра и редактирования ресурсов, роль `admin` позволяет создавать объекты `Role` и `RoleBinding`. Ее обычно выдают администраторам приложений, чтобы делегировать им обязанности по управлению пространством имен.

Эти встроенные роли служат хорошей отправной точкой. С другой стороны, их область действия можно считать слишком обширной, так как они дают доступ к большому количеству ресурсов в API-интерфейсе Kubernetes. Следуя принципу наименьших привилегий, вы можете создавать ограниченные роли, которые позволяют выполнять задачи с использованием минимального набора ресурсов и действий. Но имейте в виду, что их администрирование может потребовать накладных расходов, так как вам, возможно, придется иметь дело с уникальными ролями.



В большинстве развертываний Kubernetes клиентам обычно позволено запрашивать список всех пространств имен в кластере. Это может быть проблемой, если клиентам нельзя раскрывать информацию о существовании других пространств, так как в настоящее время этого невозможно достичь с помощью системы Kubernetes RBAC. Если у вас есть такое требование, вы должны создать для этого более высокоуровневый механизм. Пример такого механизма — ресурс Project (**https://oreil.ly/xIAT8**) от OpenShift.

Система RBAC незаменима при размещении нескольких клиентов в одном кластере. Она обеспечивает изоляцию на уровне плоскости управления, необходимую для того, чтобы не дать клиентам просматривать и изменять ресурсы друг друга. Обязательно применяйте RBAC при создании мультитенантной платформы на основе Kubernetes.

Квоты на ресурсы

Предоставляя мультитенантную платформу, вы должны убедиться в том, что каждый клиент получает адекватную долю ресурсов кластера (объем которых ограничен). В противном случае ничто не помешает амбициозному (или, возможно, злонамеренному) клиенту занять весь кластер и фактически лишить ресурсов своих соседей.

Чтобы ограничить потребление ресурсов, можно воспользоваться функцией квот из состава Kubernetes. Квоты действуют на уровне пространства имен и могут ограничивать два вида ресурсов. С одной стороны, вы можете контролировать объем вычислительных ресурсов, таких как процессор, память и хранилище, доступных пространству имен. С другой, вы можете ограничить число объектов API-интерфейса, которые можно создать в этом пространстве, включая Pod'ы, Сервисы и т. д.

Примером объектов API-интерфейса, число которых имеет смысл ограничить, являются Сервисы LoadBalancer, применяемые в облачных окружениях, так как они могут быть недешевыми.

Поскольку квоты действуют на уровне пространства имен, ваша стратегия по использованию объектов `Namespace` влияет на конфигурацию квот. Если клиенты имеют доступ к отдельному пространству, назначить квоты каждому из них довольно легко — нужно просто создать в каждом пространстве объекты `ResourceQuota`. Если же у клиентов есть доступ к нескольким пространствам имен, ситуация усложняется. В этом случае вам нужны дополнительные средства автоматизации или контроллер, чтобы обеспечить соблюдение квот в разных пространствах; эту проблему пытается решить проект Hierarchical Namespace Controller (**https://oreil.ly/PyPDK**).

Давайте рассмотрим объекты `ResourceQuota` в действии. В листинге 12.2 показан ресурс `ResourceQuota`, который позволяет пространству имен потреблять не больше одного процессора и 512 МиБ памяти.

Листинг 12.2

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: cpu-mem
  namespace: app1
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 512Mi
    limits.cpu: "1"
    limits.memory: 512Mi
```

По мере того, как Pod'ы начинают распределяться по пространству имен `app1`, им выделяется объем ресурсов, соответствующий квоте. Например, если создать Pod,

которая запрашивает 0,5 процессора и 256 Мб, следующая команда выведет обновленную квоту:

```
$ kubectl describe resourcequota cpu-mem
Name:            cpu-mem
Namespace:       app1
Resource         Used   Hard
--------         ----   ----
limits.cpu       500m   1
limits.memory    512Mi  512Mi
requests.cpu     500m   1
requests.memory  512Mi  512Mi
```

Как демонстрирует следующее сообщение об ошибке, попытки потребления ресурсов, выходящих за рамки квоты, блокируются контроллером допуска. В данном случае мы пытались потребить 2 процессора и 2 ГиБ памяти, но наш запрос был ограничен квотой:

```
$ kubectl apply -f my-app.yaml
Error from server (Forbidden):
  error when creating "my-app.yaml": pods "my-app" is forbidden:
    exceeded quota: cpu-mem,
      requested: limits.cpu=2,limits.memory=2Gi,
                 requests.cpu=2,requests.memory=2Gi,
      used: limits.cpu=0,limits.memory=0,
                 requests.cpu=0,requests.memory=0,
      limited: limits.cpu=1,limits.memory=512Mi,
                 requests.cpu=1,requests.memory=512Mi
```

Как видите, объекты ResourceQuota дают возможность контролировать то, как клиенты потребляют ресурсы кластера. Они играют ключевую роль в мультитенантных кластерах, так как благодаря им клиенты могут безопасно разделять ограниченные ресурсы.

Веб-хуки допуска

У Kubernetes есть набор встроенных контроллеров допуска, с помощью которых можно обеспечить соблюдение политики. Они служат для реализации функций объекта `ResourceQuota`, которые мы только что рассмотрели. Встроенные контроллеры помогают решать распространенные задачи, но на практике мы видим, что организациям обычно приходится расширять уровень допуска для дальнейшей изоляции клиентов и ограничения их доступа.

Проверяющие и изменяющие веб-хуки — это механизмы, которые позволяют внедрять пользовательскую логику в процесс допуска. Мы не станем углубляться в детали их реализации, так как они уже были рассмотрены в *главе 8*. Вместо этого мы возьмем некоторые сценарии использования, которые нам встречались на практике, и исследуем их в контексте мультитенантности с применением пользовательских веб-хуков допуска:

- *Стандартные метки.* С помощью проверяющих веб-хуков можно сделать так, чтобы все объекты API-интерфейса задействовали стандартный набор меток.

Например, вы можете сделать метку `owner` обязательной для всех ресурсов. Это удобно, поскольку метки позволяют запрашивать у кластера объекты и даже поддерживать такие высокоуровневые возможности, как сетевые политики и ограничения планирования.

- *Обязательные поля.* Как и в случае со стандартным набором меток, вы можете с помощью проверяющих веб-хуков допуска сделать поля определенных ресурсов обязательными. Например, вы можете требовать, чтобы все клиенты указывали поле `https` в своих ресурсах Ingress, или, например, чтобы они всегда использовали проверки готовности и работоспособности в спецификациях своих Pod'ов.
- *Ограничение возможностей.* Kubernetes имеет широкий спектр возможностей, которые иногда имеет смысл ограничить или даже отключить. Веб-хуки позволяют вам устанавливать ограничения для определенных функций. В качестве примеров можно привести отключение определенных типов Сервисов (таких как NodePort), запрет селекторов узлов, управление сетевыми именами Ingress и др.
- *Квоты на ресурсы, охватывающие разные пространства имен.* Нам встречались реальные случаи, когда организациям нужно было обеспечивать соблюдение квот на ресурсы в пределах нескольких пространств имен. Для реализации этого функционала можно создать собственный веб-хук/контроллер допуска, так как объект `ResourceQuota` из состава Kubernetes действует в пределах одного пространства имен.

В целом, веб-хуки допуска отлично подходят для того, чтобы обеспечить соблюдение пользовательской политики в мультитенантных кластерах. А появление систем управления политиками, таких как Open Policy Agent (OPA) (**https://www.openpolicyagent.org**) и Kyverno (**https://github.com/kyverno/kyverno**), делает реализацию этого подхода еще проще. Исследуйте возможность применения таких систем для изоляции клиентов в своем кластере и ограничения их прав доступа.



APIPriorityAndFairness (**https://oreil.ly/lA7jy**) — это еще один механизм Kubernetes для изоляции клиентов на уровне плоскости управления. Он защищает API-сервер от перегрузок, ограничивая число параллельных запросов, которые тот обрабатывает, в соответствии с настраиваемой политикой.

API-сервер лежит в основе функций плоскости управления. Если один клиент его перегрузит, это с большой долей вероятности существенно отразится на других. APIPriorityAndFairness может пресечь любые попытки вызвать эту перегруженность со стороны недобросовестных или дефектных клиентов API-интерфейса. Клиентские запросы сохраняются в очередь или отклоняются в соответствии с настраиваемой политикой.

APIPriorityAndFairness является относительно новой функцией. На момент написания этих строк она находится в состоянии alpha-тестирования, и мы еще не видели, чтобы кто-то применил ее в реальных условиях. Мы советуем повременить с ее включением, если только у вас нет веской причины для этого. Но, даже если вам нужна такая возможность, подумайте о том, что вариант с несколькими кластерами может быть более прост в реализации.

Запросы и лимиты на ресурсы

Kubernetes планирует выполнение приложений в разделяемом пуле узлов кластера. Обычно приложения разных клиентов развертываются на одном и том же узле, совместно используя его ресурсы. Справедливое разделение ресурсов — самый важный аспект эксплуатации мультитенантной платформы. Без этого клиенты, размещенные на одном узле, могут отрицательно влиять друг на друга.

Запросы и лимиты на ресурсы — это механизмы Kubernetes, которые изолируют клиентов друг от друга в контексте вычислительных ресурсов. Запросы ресурсов в целом удовлетворяются на уровне планировщика Kubernetes (и, как вы увидите позже, запросы ресурсов процессора отражаются во время выполнения). Для сравнения, лимиты на ресурсы реализуются на уровне узла с помощью контрольных групп (cgroups) и планировщика CFS (Completely Fair Scheduler), входящих в состав Linux.



Запросы и лимиты обеспечивают адекватную степень изоляции для приложений, однако следует понимать, что эта изоляция не настолько строгая, как та, которую предоставляет гипервизор. Полное устранение проявлений "шумного соседа" в контейнерных окружениях может быть непростой задачей. Обязательно поэкспериментируйте и убедитесь в том, что вы понимаете последствия, которые влечет за собой размещение нескольких высоконагруженных приложений на отдельно взятом узле Kubernetes.

Помимо обеспечения изоляции ресурсов, запросы и лимиты на ресурсы определяют класс качества обслуживания (англ. Quality of Service или QoS) Pod'а. Класс QoS играет важную роль, диктуя порядок, в котором kubelet выгружает Pod'ы в условиях нехватки ресурсов на узле. Kubernetes предлагает следующие классы QoS:

- *Guaranteed* — относится к Pod'ам, у которых лимиты на процессор и память равны соответствующим запросам. Это касается всех контейнеров. Kubelet редко выгружает гарантированные Pod'ы.
- *Burstable* — относится к Pod'ам, которые не соответствуют классу Guaranteed и имеют как минимум один контейнер с запросами процессора или памяти. Kubelet выгружает Pod'ы этого класса в зависимости от того, насколько потребляемые ими ресурсы превышают значения, указанные в их запросах. Чем сильнее потребление выходит за рамки запросов, тем выше вероятность того, что Pod'ы будут выгружены.
- *BestEffort* — относится к Pod'ам, у которых нет запросов или лимитов на процессор или память. Такие Pod'ы работают без каких-либо гарантий, и в случае необходимости kubelet выгрузит их первыми.



Выгрузка Pod'ов — сложный процесс. Принимая решение о выгрузке, kubelet учитывает не только классы QoS, но и приоритеты Pod'ов. В документации Kubernetes есть замечательная статья, в которой подробнейшим образом описываются возможные действия при "исчерпании ресурсов" (**https://oreil.ly/LuCD9**).

Теперь мы знаем, что запросы и лимиты на ресурсы обеспечивают изоляцию клиентов и определяют класс QoS Pod'а. Давайте подробно рассмотрим эти механиз-

мы. Несмотря на то, что запросы и лимиты Kubernetes поддерживают разные ресурсы, в центре нашего внимания будут находиться процессор и память, которые требуются для выполнения любых приложений. Для начала обсудим запросы и лимиты на память.

Каждый контейнер в Pod'е может указывать запросы и лимиты на ресурсы. Планировщик слагает запросы памяти отдельных контейнеров, чтобы получить общий запрос Pod'а. Получив эту информацию, планировщик находит узел с достаточным объемом памяти для размещения этого Pod'а. Если ни один из узлов кластера не удовлетворяет заявленным требованиям, Pod остается в состоянии ожидания. В случае развертывания контейнеры Pod'а гарантированно получают запрошенную память.

Запрос памяти отражает минимальный гарантированный объем этого ресурса. Но Pod может потреблять и дополнительные ресурсы, если они доступны на узле. Это проблематично, так как планировщик может назначить память, занимаемую Pod'ом, другим приложениям или клиентам. Если на узле будет развернут новый Pod, он может бороться за память с уже существующим. Чтобы удовлетворить запросы обоих Pod'ов, планировщик удалит тот из них, у которого потребляемый объем памяти превышает запрошенный.

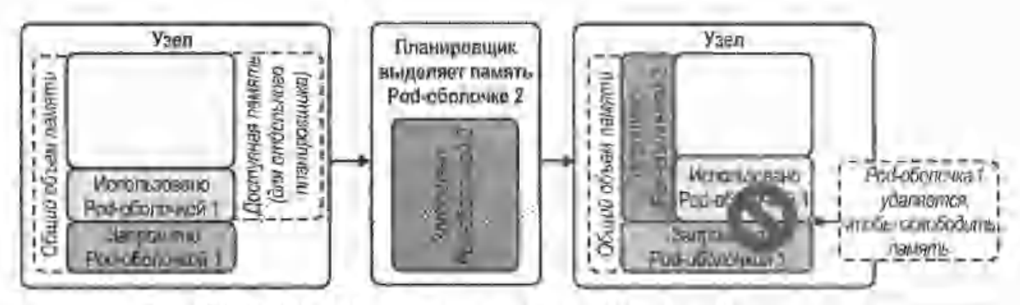


Рис. 12.3. Pod, потребляемая память которого превышает запрошенную, удаляется, чтобы освободить память для нового Pod'а

Чтобы контролировать объем памяти, который могут потреблять клиенты, мы должны указать соответствующие лимиты для приложений. В результате приложению будет доступно не больше того, что мы ему выделили. Если оно попытается потребить память за рамками лимита, это приведет к его принудительному завершению. Дело в том, что память — это нерегулируемый ресурс, ее объем нельзя изменять на лету. Если узел испытывает нехватку памяти, процесс должен быть принудительно завершен. В листинге 12.3 показан контейнер, который был остановлен из-за исчерпания доступной памяти (англ. Out-of-Memory Killed или OOMKilled). Обратите внимание на поле `Reason` (причина) в разделе `Last State` (последнее состояние).

Листинг 12.3

```
$ kubectl describe pod memory
Name:         memory
```

```
Namespace:    default
Priority:     0
... <опущено> ...
Containers:
  stress:
    ... <опущено> ...
    Last State:     Terminated
      Reason:       OOMKilled
      Exit Code:    1
      Started:      Fri, 23 Oct 2020 10:11:51 -0400
      Finished:     Fri, 23 Oct 2020 10:11:56 -0400
    Ready:          True
    Restart Count:  1
    Limits:
      memory:  100Mi
    Requests:
      memory:     100Mi
```

Когда мы участвуем в реальных проектах, нас часто спрашивают, стоит ли позволять клиентам устанавливать лимиты на память выше запросов. Должны ли узлы "обещать" больше памяти, чем у них есть на самом деле? Ответ на этот вопрос сводится к балансу между плотностью и стабильностью. Выделяя слишком много памяти, вы делаете узел более заполненным, но менее стабильным. Как мы уже видели, в случае нехватки памяти приложение, превысившее свои запросы по ресурсам, принудительно завершается. В большинстве случаев мы советуем разработчикам платформ не предоставлять память, которой у узла физически нет, так как стабильность для них обычно важнее, чем возможность плотно заполнить свои узлы. Это особенно касается кластеров, в которых размещаются приложения в режиме эксплуатации.

Итак, мы обсудили запросы и лимиты на память. Теперь перейдем к процессору. В отличие от памяти, центральный процессор — регулируемый ресурс. В случае его нехватки выполнение процессов можно замедлить. По этой причине запросы и лимиты на процессор оказываются несколько сложнее, чем на память.

Запросы и лимиты на процессор указываются в единицах измерения CPU. В большинстве случаев 1 CPU является эквивалентом одного процессорного ядра. Запросы и лимиты могут быть дробными (например, 0,5 CPU), и их можно выражать в милли-CPU, указав суффикс "m". 1 CPU равен 1000m CPU.

Если у контейнеров Pod'а заданы запросы процессора, планировщик ищет для нее узел с достаточным количеством вычислительных ресурсов. После развертывания kubelet преобразует запрошенные единицы измерения CPU в доли CPU. Доли CPU (англ. CPU shares) — это механизм в ядре Linux, который выделяет процессорное время группам cgroup (т. е. процессам внутри cgroup). Вот ключевые аспекты этого механизма, о которых следует помнить:

- Доли CPU являются относительными. 1 000 долей CPU не означает одно или тысячу процессорных ядер. Вместо этого вычислительная емкость процессора

распределяется между всеми группами cgroup в соответствии с их долей. Возьмем, к примеру, два процесса в разных группах. Если у процесса 1 (П1) есть 2 000 долей, а у процесса 2 (П2) их 1 000, то П1 получит вдвое больше процессорного времени, чем П2.

- Доли CPU вступают в силу только в случае нехватки вычислительных ресурсов. Если процессор занят не полностью, скорость выполнения процессов не ограничивается, что позволяет им потреблять дополнительные циклы процессора. Возвращаясь к предыдущему примеру, П1 получит в два раза больше процессорного времени, чем П2, только если процессор занят на 100 %.

Доли CPU (запросы CPU) обеспечивают изоляцию вычислительных ресурсов, необходимую для размещения нескольких клиентов на одном и том же узле. Емкость процессора распределяется между клиентами в соответствии с запросами, которые они объявили. Благодаря этому они не могут лишить своих соседей процессорного времени.

Лимиты на процессор работают иначе. Они устанавливают максимальное количество процессорного времени, доступное контейнерам. Для реализации лимитов на процессор Kubernetes использует механизм управления пропускной способностью, встроенный в планировщик CFS (Completely Fair Scheduler). Этот механизм предусматривает наличие периодов времени, чтобы ограничить потребление процессора. Каждый контейнер получает квоту в рамках настраиваемого периода. Квота определяет, сколько процессорного времени может потребляться за каждый период. Если контейнер исчерпает квоту, его производительность ограничивается до окончания периода.

Kubernetes устанавливает по умолчанию период длительностью 100 мс. Как видно на рис. 12.4, контейнер с лимитом 0,5 CPU получает 50 мс процессорного времени каждые 100 мс. Контейнер с лимитом 3 CPU получает 300 мс процессорного времени на каждом интервале длительностью 100 мс, что фактически позволяет ему потреблять 3 CPU на протяжении 100 мс.

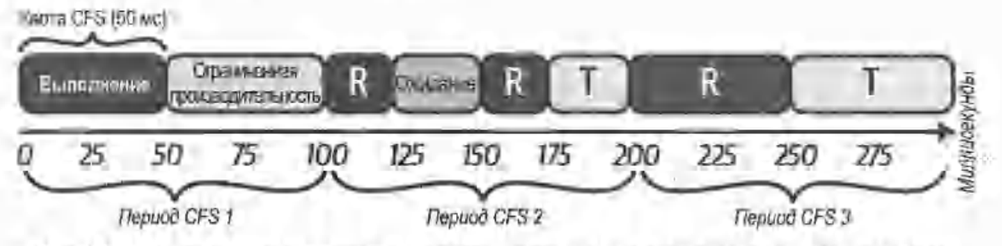


Рис. 12.4. Потребление ресурсов процессора и ограничение производительности процесса, который выполняется в cgroup и имеет период CFS длиной 100 мс и квоту процессорного времени в размере 50 мс

Ввиду своей природы лимиты на процессор могут иногда приводить к неожиданному поведению или непредвиденному ограничению производительности. Это обычно происходит с многопоточными приложениями, которые могут израсходовать всю квоту на начальных этапах выполнения. Например, контейнер с лимитом

1 CPU будет получать 100 мс процессорного времени каждые 100 мс. Если предположить, что у него есть 5 потоков, потребляющих вычислительные ресурсы, то он исчерпает свою квоту за 20 мс, и оставшиеся 80 мс будет находиться в состоянии ограниченной производительности. Такая ситуация изображена на рис. 12.5.

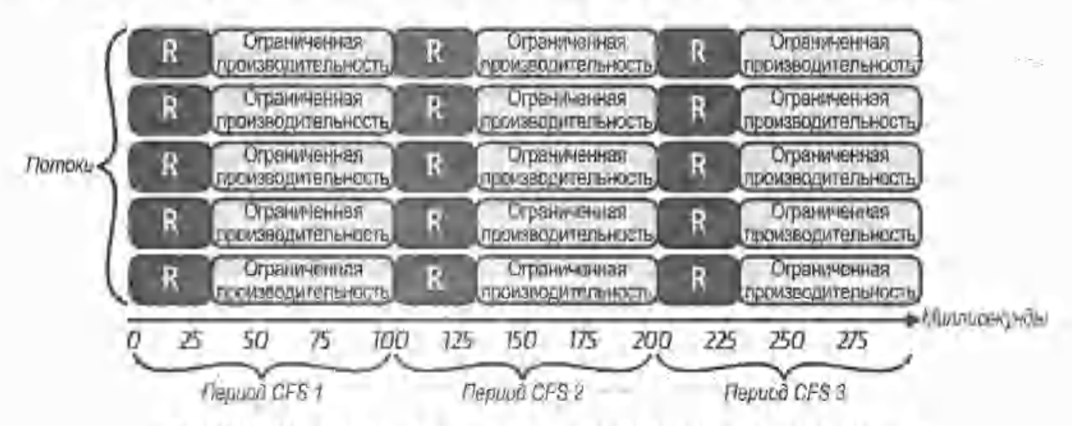


Рис. 12.5. Многопоточное приложение исчерпало всю квоту на процессор за первые 20 мс 100-миллисекундного периода

Применение лимитов на процессор позволяет минимизировать колебания производительности приложения, особенно при выполнении нескольких реплик на разных узлах. Эти колебания возникают из-за того, что реплики, у которых нет лимитов, могут иметь всплески производительности и потреблять незанятые циклы процессора, которые *могут* быть доступны в разные моменты времени. Указывая лимиты на процессор, равные соответствующим запросам, вы избавляетесь от этих колебаний, так как приложения получают ровно столько ресурсов, сколько они запросили (Google и IBM опубликовали замечательное исследование, в котором подробно анализируется механизм регуляции пропускной способности CFS: **https://oreil.ly/39Pu7**). Аналогичным образом лимиты на процессор играют ключевую роль в сравнительном тестировании производительности. Без лимитов ваши тесты имели бы неубедительные результаты, так как вычислительные ресурсы, доступные для ваших приложений, зависели бы от того, на каких узлах те развертываются и сколько свободного процессорного времени там доступно.

Если вашим приложениям нужен предсказуемый доступ к вычислительным ресурсам (например, если они чувствительны к задержкам), то целесообразно установить лимиты на процессор, равные соответствующим запросам. В остальном ограничение максимального количества процессорных циклов является излишним. Когда ресурсы процессора на узле исчерпываются, механизм CPU shares гарантирует, что приложения получат справедливую долю ресурсов в соответствии с запросами их контейнеров. Если же у узла есть свободные вычислительные ресурсы, они не простаивают впустую, так как приложения потребляют их по мере необходимости.

В ядре Linux есть программная ошибка (**https://oreil.ly/EPWrm**), которая ограничивает производительность контейнеров без необходимости. Это сильно влияет на

приложения, чувствительные к задержкам, такие как веб-сервисы. Для решения указанной проблемы пользователи Kubernetes прибегают к разным обходным путям:

- удалению лимитов на процессор из спецификации Pod;
- отключению соблюдения лимитов на процессор за счет передачи kubelet флага `--cpu-cfs-quota=false`;
- сокращению периода CFS до 5–10 мс за счет передачи kubelet флага `--cpu-cfs-quota-period`.

Возможно, вам и не придется использовать эти обходные решения, так как данная ошибка была исправлена (**https://oreil.ly/xekUx**) в версии 5.4 ядра Linux с переносом исправления в версии 4.14.154+, 4.19.84+ и 5.3.9+. Если вам нужно обеспечить соблюдение лимитов на процессор, обновите свое ядро Linux, чтобы избежать этого дефекта.

Сетевые политики

В большинстве развертываний Kubernetes исходит из того, что Pod'ы, работающие на платформе, могут взаимодействовать друг с другом. Как можно догадаться, такой подход будет проблематичным для мультитенантных кластеров, в которых есть необходимость в изоляции клиентов на уровне сети. Чтобы гарантировать такую изоляцию, можно воспользоваться API-интерфейсом NetworkPolicy.

Мы уже рассматривали сетевые политики в *главе 5*, где обсуждалась роль, которую играют в их соблюдении подключаемые модули CNI (Container Networking Interface — интерфейс управления сетью контейнеров). В этом разделе мы поговорим о модели *отклонения всех запросов по умолчанию* — распространенном подходе к использованию NetworkPolicy, особенно в мультитенантных кластерах.

Вы, как администратор платформы, можете установить сетевую политику отклонения всех запросов по умолчанию для всего кластера. Таким образом вы примете самые жесткие меры безопасности и изоляции: клиенты становятся полностью изолированными друг от друга сразу после размещения на платформе. Более того, эта модель подталкивает клиентов к объявлению сетевых взаимодействий своих приложений, что усиливает их сетевую безопасность.

Что касается реализации политики отклонения всех запросов по умолчанию, вы можете выбрать один из двух путей, каждый из которых имеет свои плюсы и минусы. Первый путь состоит в использовании API-интерфейса NetworkPolicy, доступного в Kubernetes. Поскольку этот интерфейс является одним из основных, данная реализация совместима с разными подключаемыми модулями CNI. Но, учитывая, что объект `NetworkPolicy` действует в рамках одного пространства имен, вам придется создавать и администрировать множество ресурсов NetworkPolicy, по одному в каждом пространстве. Кроме того, клиентам нужно разрешение для создания объектов `NetworkPolicy`, поэтому вы должны реализовать дополнительные механизмы (обычно в виде веб-хуков допуска, как обсуждалось ранее), чтобы клиенты не могли удалить эту политику. В листинге 12.4 показан объект `NetworkPolicy`, отклоняю-

щий все запросы по умолчанию. Пустой Pod-селектор выбирает все Pod'ы в пространстве имен.

Листинг 12.4

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: tenant-a
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

Альтернативный подход заключается в применении CRD (Custom Resource Definitions — определений пользовательских ресурсов), ориентированных на конкретный модуль CNI. Некоторые модули CNI, такие как Antrea, Calico и Cilium, предоставляют CRD, с помощью которой вы можете указать "глобальную" сетевую политику (уровня кластера). Эти пользовательские ресурсы помогают упростить реализацию и администрирование политики отклонения всех запросов по умолчанию, но привязывают вас к определенному модулю CNI. В листинге 12.5 приведен пример пользовательского ресурса `GlobalNetworkPolicy` из состава Calico, который реализует эту политику.

Листинг 12.5

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: default-deny
spec:
  selector: all()
  types:
  - Ingress
  - Egress
```



Обычно реализации сетевой политики отклонения всех запросов по умолчанию делают исключения для инфраструктурного сетевого трафика, такого как обращения к DNS-серверу кластера. Соответственно, они не применяются к пространству имен kube-system и любому другому пространству системного уровня, чтобы не нарушать работу платформы. Фрагменты YAML-кода, представленные выше, этого не учитывают.

Как это часто бывает, выбор между встроенным объектом `NetworkPolicy` и CRD сводится к поиску баланса между переносимостью и простотой. Как показывает наш опыт, простота, которой удается достичь за счет использования пользовательского

ресурса для конкретного модуля CNI, того стоит, учитывая, что переход с одного модуля на другой происходит нечасто. С другой стороны, в будущем вам, возможно, не придется принимать это решение, поскольку специальная группа Kubernetes по вопросам сетевых технологий (sig-network) рассматривает вопрос о расширении API-интерфейсов NetworkPolicy для поддержки общекластерных сетевых политик (**https://oreil.ly/jVP_f**).

Когда в кластере установлена политика отклонения всех запросов по умолчанию, клиенты сами должны создавать для себя лазейки в структуре сети, чтобы обеспечить работу своих приложений. Для этого они используют ресурс NetworkPolicy, в котором указываются правила для входящего и исходящего трафика приложений. Например, в листинге 12.6 показан ресурс `NetworkPolicy`, который можно применить к веб-сервису. Он разрешает прием входящего трафика от веб-клиентов (Ingress) и отправку исходящего трафика к базе данных.

Листинг 12.6

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: webservice
  namespace: reservations
spec:
  podSelector:
    matchLabels:
      role: webservice
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 8080
  egress:
  - to:
    - podSelector:
        role: database
    ports:
    - protocol: TCP
      port: 3306
```

Гарантия соблюдения сетевой политики, отклоняющей все запросы по умолчанию, — это один из важнейших элементов механизма изоляции. Мы настоятельно

советуем применять этот подход тем, кто создает платформы на основе Kubernetes, особенно если вы планируете размещение множества клиентов.

Политики безопасности Pod

Политики безопасности Pod (англ. Pod Security Policies или PSP) — еще один важный механизм, благодаря которому клиенты могут безопасно сосуществовать в одном кластере. PSP управляют важными параметрами безопасности Pod'ов на этапе выполнения, такими как возможность получения повышенных привилегий, доступ к томам хоста, привязка с сетевому интерфейсу хоста и т. д. Без PSP (или аналогичного механизма соблюдения политик) клиенты могли бы делать в кластере практически все, что им хочется.

Kubernetes обеспечивает соблюдение большинства ограничений, реализованных с помощью PSP, посредством контроллера допуска (соблюдения правила, которое требует, чтобы пользователь не был администратором, иногда обеспечивается агентом kubelet, который проверяет пользователя контейнера на этапе выполнения после скачивания образа). Когда контроллер допуска включен, все попытки создания Pod блокируются, если только они не разрешены политикой PSP. В листинге 12.7 показана запретительная PSP, которую мы обычно применяем *по умолчанию* в мультитенантных кластерах.

Листинг 12.7. Пример запретительной политики PodSecurityPolicy

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: default
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: |
      'docker/default,runtime/default'
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName: 'runtime/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName: 'runtime/default'
spec:
  privileged: false ❶
  allowPrivilegeEscalation: false
  requiredDropCapabilities:
    - ALL
  volumes: ❷
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    - 'persistentVolumeClaim'
```

```
  hostNetwork: false ❸
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: 'MustRunAsNonRoot' ❹
  seLinux:
    rule: 'RunAsAny' ❺
  supplementalGroups: ❻
    rule: 'MustRunAs'
    ranges:
      - min: 1
        max: 65535
  fsGroup: ❼
    rule: 'MustRunAs'
    ranges:
      - min: 1
        max: 65535
  readOnlyRootFilesystem: false
```

❶ Запрещаем привилегированные контейнеры.

❷ Ограничиваем типы томов, доступные Pod'ам.

❸ Не даем Pod'ам привязываться к сетевому стеку их хостов.

❹ Делаем так, чтобы контейнеры выполнялись от имени обычного пользователя.

❺ Эта политика подразумевает, что на узлах используется AppArmor, а не SELinux.

❻ Перечисляем ID групп, которые позволены контейнерам. ID корневой группы (0) запрещен.

❼ Определяем идентификаторы групп, применяемых к томам. ID корневой группы (0) запрещен.

Одного наличия политики PSP, которая допускает Pod, недостаточно для того, чтобы этот Pod была допущен. В дополнение к этому он должен быть авторизован для *использования* PSP. Авторизация PSP проводится с помощью RBAC. Pod'ы могут задействовать PSP, если это позволено делать их служебной учетной записи (англ. Service Account или SA). Однако учитывая, что Pod'ы редко создаются пользователями кластера, применение служебной учетной записи для авторизации PSP является более распространенным подходом. В листинге 12.8 показаны объекты `Role` и `RoleBinding`, которые авторизуют SA для использования определенной политики PSP с названием `sample-psp`.

Листинг 12.8

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: sample-psp
```

```
rules:
- apiGroups: ['policy']
  resources: ['podsecuritypolicies']
  resourceNames: ['sample-psp']
  verbs: ['use']
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: sample-psp
subjects:
- kind: ServiceAccount
  name: my-app
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: sample-psp
```

В большинстве случаев администраторы платформы отвечают за создание и администрирование политик PSP, а также за предоставление их клиентам. Разрабатывая свои политики, всегда соблюдайте принцип наименьших привилегий. Делайте доступным минимальный набор прав и возможностей, необходимый Pod'ам для выполнения их работы. Мы обычно рекомендуем начинать с создания следующих политик:

- *Default* — политика по умолчанию доступна всем клиентам кластера. Она должна быть запретительной и блокировать все привилегированные операции, делать недоступными любые возможности Linux, запрещать выполнение от имени root и т. д. (YAML-файл с определением этой политики показан в листинге 12.7). Чтобы эта политика применялась по умолчанию, вы можете сделать так, чтобы она проводила авторизацию всех Pod'ов в кластере. Для этого используйте объекты `ClusterRole` и `ClusterRoleBinding`.
- *Kube-system* — эта политика предназначена для системных компонентов, которые находятся в пространстве имен kube-system. Учитывая характер этих компонентов, данная политика должна быть менее запретительной, чем default. Например, она должна позволять Pod'ам подключать тома hostPath и выполнять код от имени root. В отличие от политики по умолчанию, для авторизации RBAC предназначен объект `RoleBinding`, применяемый ко всем служебным учетным записям в пространстве имен kube-system.
- *Networking* — политика ориентирована на сетевые компоненты кластера, такие как подключаемые модули CNI. Этим Pod'ам нужно еще больше привилегий для работы с сетевым стеком узлов кластера. Чтобы применять эту политику только к сетевым Pod'ам, создайте объект `RoleBinding`, который авторизует ее использование только соответствующими служебными учетными записями.

Наличие этих политик позволяет клиентам развертывать в кластере непривилегированные приложения. Если приложению требуются дополнительные привилегии,

вам нужно определить, приемлете ли вы риск его выполнения в том же кластере. Если да, то создайте отдельную политику, рассчитанную специально для него. Выдайте ему привилегии, которые оно требует, и сделайте его служебную учетную запись единственной, которой позволено использовать эту политику PSP.

Политики PSP являются важным механизмом, обеспечивающим соблюдение ограничений в мультитенантной платформе. Они определяют, что клиенты могут и не могут делать на этапе выполнения, разделяя узлы с другими. При создании своей платформы вы должны обеспечить изоляцию клиентов и защитить их друг от друга с помощью PSP.



В сообществе Kubernetes обсуждается (**https://oreil.ly/ayN8j**) возможность удаления API-интерфейса и контроллера допуска PodSecurityPolicy из основного проекта. Если это произойдет, аналогичные возможности можно будет реализовать с помощью таких систем управления политиками, как Open Policy Agent (**https://oreil.ly/wrz23**) или Kyverno (**https://oreil.ly/v7C2H**).

Сервисы мультитенантных платформ

Изолировать можно не только плоскость управления Kubernetes и плоскость рабочих заданий, но и различные сервисы, которые предлагает ваша платформа, включая журналирование, мониторинг, управление входящим трафиком и т. д. Важным определяющим фактором в реализации этого механизма являются технологии, с помощью которых предоставляется сервис. Некоторые из них могут иметь встроенную поддержку мультитенантности, что существенно упрощает процесс реализации.

Еще один важный фактор состоит в том, *нужно ли* вам изолировать клиентов на этом уровне. Хотите ли вы, чтобы клиенты не могли просматривать журнальные записи и метрики друг друга? Следует ли им разрешать свободно обнаруживать чужие сервисы посредством DNS? Могут ли они использовать общий маршрут для входящего трафика? Ответив на эти и аналогичные вопросы, вы сможете прояснить свои требования. В итоге все сводится к степени доверия между клиентами, которые размещаются на вашей платформе.

Помогая разработчикам платформ, мы часто имеем дело с мультитенантным мониторингом на основе Prometheus. У Prometheus нет встроенной поддержки мультитенантности. Метрики потребляются и хранятся в единой базе данных временных рядов, доступной любому, у кого есть доступ к HTTP-пути Prometheus. Иными словами, если экземпляр Prometheus собирает метрики разных клиентов, у нас нет возможности запретить этим клиентам просматривать данные друг друга. Для решения этой проблемы мы должны развертывать для каждого из них отдельные экземпляры Prometheus.

В таких ситуациях мы обычно используем prometheus-operator (**https://oreil.ly/j38-Q**). Как уже обсуждалось в *главе 9*, prometheus-operator позволяет развертывать и администрировать сразу несколько копий Prometheus с помощью определений пользовательских ресурсов. Благодаря этому вы можете предоставлять в рамках своей платформы сервис мониторинга, способный безопасно обслуживать множество

клиентов. Клиенты полностью изолированы друг от друга, так как каждый из них получает отдельный стек мониторинга, состоящий из Prometheus, Grafana, Alertmanager и т. д.

В зависимости от того, какие возможности должна предоставлять ваша платформа, вы можете либо позволить клиентам развертывать свои собственные экземпляры Prometheus с помощью prometheus-operator, либо создавать их при добавлении новых клиентов. Мы рекомендуем второй вариант, если у разработчиков платформы есть такая возможность, это освободит клиентов от лишних обязанностей и повысит удобство использования платформы.

Централизованное журналирование — это еще один сервис платформы, который можно реализовать с учетом мультитенантности. Обычно это подразумевает отправку журнальных записей каждого клиента отдельному компоненту или хранилищу. У большинства маршрутизаторов журнальных записей есть механизмы, позволяющие реализовать мультитенантное решение.

Если вы используете инструменты Fluentd и Fluent Bit, вам доступна маршрутизация на основе тегов. В листинге 12.9 приведен пример выходной конфигурации Fluent Bit, которая направляет журнальные записи Элис (принадлежащие Pod'ам в пространстве имен `alice-ns`) к одному компоненту, а журнальные записи Боба (принадлежащие Pod'ам в пространстве имен `bob-ns`) — к другому.

Листинг 12.9

```
[OUTPUT]
    Name            es
    Match           kube.var.log.containers.**alice-ns**.log
    Host            alice.es.internal.cloud.example.com
    Port            ${FLUENT_ELASTICSEARCH_PORT}
    Logstash_Format On
    Replace_Dots    On
    Retry_Limit     False

[OUTPUT]
    Name            es
    Match           kube.var.log.containers.**bob-ns**.log
    Host            bob.es.internal.cloud.example.com
    Port            ${FLUENT_ELASTICSEARCH_PORT}
    Logstash_Format On
    Replace_Dots    On
    Retry_Limit     False
```

Помимо изоляции журнальных записей на стороне сервера, вы также можете реализовать ограничение пропускной способности или производительности, чтобы отдельно взятый клиент не мог занять всю инфраструктуру маршрутизации журналов своими запросами. У инструментов Fluentd и у Fluent Bit есть подключаемые модули, которые могут обеспечить соблюдение таких лимитов. Наконец, если у вас

есть более сложные рабочие сценарии, такие как предоставление доступа к конфигурации средств журналирования в виде пользовательского ресурса Kubernetes, то вы можете реализовать их поддержку с помощью logging-operator.

Иногда разработчики не уделяют должного внимания вопросам мультитенантности на уровне сервисов платформы. Создавая собственную мультитенантную платформу, взвешивайте свои требования и их влияние на то, какие сервисы вы хотите предоставлять. В некоторых случаях это может влиять на выбор подходов и инструментов, которые будут лежать в основе вашей платформы.

Резюме

Размещение приложений от разных клиентов — важный аспект, который необходимо учитывать при создании платформы на основе Kubernetes. С одной стороны, вы можете предоставлять однотенантные кластеры для каждого клиента своей платформы. Этот подход осуществим на практике, но мы обсудили его недостатки, такие как накладные расходы, связанные с ресурсами и администрированием. В качестве альтернативы можно воспользоваться мультитенантными кластерами, в которых множество клиентов могут иметь общие плоскости управления, рабочих заданий и сервисы платформы.

Размещая разных клиентов на одном кластере, вы должны обеспечить их изоляцию, чтобы они не могли влиять друг на друга отрицательным образом. Как уже обсуждалось, это можно сделать на основе пространств имен. Мы рассмотрели множество механизмов изоляции, доступных в Kubernetes, которые позволяют создать мультитенантную платформу. Такие механизмы доступны на разных уровнях, в основном в плоскости управления, плоскости рабочих заданий и в сервисах платформы.

К механизмам изоляции плоскости управления относятся RBAC (для ограничения действий клиентов), квоты на ресурсы (для распределения ресурсов кластера) и веб-хуки доступа (для соблюдения политики). В плоскости рабочих заданий клиентов можно разделять с помощью запросов и лимитов на ресурсы (чтобы обеспечить справедливое распределение ресурсов узла), сетевых политик (чтобы разделить Pod-сеть на сегменты) и Pod Security Policies (чтобы ограничить возможности Pod'ов). Наконец, что касается сервисов платформы, то вы можете использовать разные технологии для реализации мультитенантных предложений. В качестве примеров таких сервисов, в которые можно встроить поддержку множества клиентов, мы рассмотрели мониторинг и централизованное журналирование.

ГЛАВА 13

Автоматическое масштабирование

Возможность автоматического масштабирования производительности приложений — одно из важных преимуществ облачно-ориентированных систем. Если у вас есть приложения, нагрузка на которые существенно колеблется, то автомасштабирование может снизить денежные расходы и облегчить администрирование. Автомасштабирование — это процесс увеличения или уменьшения емкости наших приложений без участия со стороны человека. Все начинается с определения момента, когда необходимо изменить емкость приложения. Это подразумевает изменение параметров в зависимости от метрик. В итоге получаются системы, расширяющие и сокращающие объем ресурсов, доступный приложению, в соответствии с той работой, которую оно должно выполнить.

Несмотря на большую пользу, которую может принести автомасштабирование, необходимо понимать, когда его *не* стоит использовать. Автомасштабирование усложняет управление приложениями. Помимо начальной конфигурации, вам, скорее всего, придется пересмотреть и откорректировать параметры свих механизмов масштабирования. Поэтому, если ресурсоемкость приложения не меняется заметно, вы вполне можете выделить ему ресурсы, рассчитанные на максимальный объем трафика, с которым оно способно справиться. Если же нагрузка на приложение изменяется в предсказуемые периоды времени, ручное регулирование его емкости может оказаться достаточно легким, и вложения в автомасштабирование не оправдают себя. Любую технологию следует внедрять только в случае, если долгосрочная выгода перевешивает начальную конфигурацию и обслуживание системы.

Мы разделим тему автоматического масштабирования на две общие категории:

- *Автомасштабирование приложений* — автоматическое управление емкостью отдельных приложений.
- *Автомасштабирование кластера* — автоматическое управление емкостью платформы, на которой развернуты приложения.

В ходе обсуждения этих подходов помните об основных факторах, побуждающих к применению автомасштабирования:

- *Управление финансовыми расходами* — в первую очередь относится к тем, кто арендует серверы у общедоступного облачного провайдера или платит за внутреннюю виртуальную инфраструктуру. Автомасштабирование кластеров дает возможность динамически регулировать число серверов, за которые вы платите. Чтобы сделать свою инфраструктуру настолько эластичной, вам придется использовать автомасштабирование приложений для управления емкостью соответствующих приложений внутри кластера.

- *Управление емкостью* возможно, если у вас есть набор статических инфраструктурных компонентов. В этом случае автомасштабирование даст вам возможность динамически управлять их фиксированной емкостью. Например, у приложений, предоставляющих услуги конечным пользователям, зачастую есть дни недели и время суток, когда они испытывают пиковую нагрузку. Автомасштабирование позволяет приложениям при необходимости динамически увеличивать свою емкость и потреблять бо́льшие объемы ресурсов кластера. Оно также осуществляет освобождение потребляемых ресурсов для других приложений. Возможно, у вас есть приложения с пакетной обработкой, которые могут воспользоваться незанятыми вычислительными ресурсами во внерабочее время. Автомасштабирование может существенно облегчить управление емкостью вычислительной инфраструктуры, поскольку число серверов, входящих в состав кластера, регулируется без человеческого вмешательства.

Автомасштабирование подходит для приложений, у которых колеблются нагрузка и объем трафика. Без него у вас остается два варианта:

- Постоянно завышать емкость своего приложения, что выльется в дополнительные расходы для вашей компании.
- Обращаться к инженерам для выполнения операций масштабирования вручную, что сделает эксплуатацию более трудоемкой.

В этой главе мы сначала поговорим о том, как подходить к автомасштабированию и как проектировать программное обеспечение с расчетом на эти системы. Затем мы подробно рассмотрим конкретные системы, которые пригодны для автоматического масштабирования приложений на платформах, основанных на Kubernetes. Это будет включать в себя горизонтальное и вертикальное автомасштабирование, в том числе метрики, с помощью которых следует инициировать соответствующие события. Мы также затронем тему масштабирования приложений пропорционально самому кластеру и приведем пример создания собственного механизма автомасштабирования. В конце, в разделе "Автомасштабирование кластера", речь пойдет о масштабировании самой платформы, которое позволит нам реагировать на существенные изменения в потреблении ресурсов размещенными на ней приложениями.

Виды масштабирования

В сфере разработки программного обеспечения масштабирование делится на две категории:

- *Горизонтальное масштабирование* — изменение числа копий (реплик) приложения. Речь идет либо о числе Pod'ов определенного приложения, либо о числе узлов в кластере, на которых размещаются приложения. В дальнейшем при упоминании увеличения или уменьшения числа Pod'ов или узлов в контексте горизонтального масштабирования мы будем употреблять термины "расширение" и "сокращение" масштаба.

♦ *Вертикальное масштабирование* — изменение количества ресурсов отдельно взятого экземпляра. В случае с приложением это означает изменение запросов и/или лимитов на ресурсы для его контейнеров. Если речь об узлах кластера, это обычно подразумевает изменение количества доступных ресурсов процессора и памяти. В дальнейшем при упоминании вертикального масштабирования мы будет употреблять термины "повышение" и "понижение" масштаба.

В системах, которые нуждаются в динамическом масштабировании (т. е. испытывающих частые и существенные колебания нагрузки), по возможности следует отдавать предпочтение горизонтальному масштабированию. Вертикальное масштабирование ограничено самым производительным сервером, который вам доступен. К тому же, такое "вертикальное" повышение емкости требует перезапуска приложения. Даже в виртуальных окружениях, поддерживающих динамическое масштабирование, Pod'ы приходится перезапускать, так как запросы и лимиты на ресурсы в настоящее время нельзя обновлять на лету. Для сравнения, при горизонтальном масштабировании экземпляры приложения не нуждаются в перезапуске, а динамическое увеличение емкости происходит за счет добавления реплик.

Архитектура приложения

Вопрос автомасштабирования особенно важен для сервис-ориентированных систем. Одно из преимуществ разделения приложения на отдельные компоненты — возможность их независимого масштабирования. Мы занимались этим в многоуровневых архитектурах задолго до появления облачных систем. Отделение веб-приложений от реляционных баз данных и их независимое масштабирование стало общепринятой практикой. В микросервисных архитектурах можно пойти еще дальше. Например, у веб-сайта предприятия может быть два сервиса, один из которых обслуживает его Интернет-магазин, а другой отвечает за статьи в блоге. Во время рекламных акций Интернет-магазин можно масштабировать, а сервис блога можно оставить без изменений, так как на него эти акции не влияют.

Имея возможность масштабировать сервисы независимо друг от друга, вы можете более эффективно задействовать инфраструктуру, которую используют ваши приложения. Но в то же время повышаются накладные расходы, связанные с масштабированием множества отдельных приложений. Автоматизация этого процесса становится очень важной. На определенном этапе без нее уже не обойтись.

Автомасштабирование хорошо подходит для небольших, легковесных приложений с крошечными образами и коротким временем запуска. Если скачивание образа на заданный узел и запуск соответствующего контейнера происходит быстро, приложение может оперативно реагировать на события масштабирования. Это существенно облегчает изменение емкости. А вот приложения с образами, занимающими гигабайт, и скриптами запуска, время выполнения которых исчисляется минутами, куда менее приспособлены к реагированию на изменения нагрузки. Такие приложения не подходят для автомасштабирования. Помните об этом при проектировании и реализации своих приложений.

Также необходимо понимать, что автомасштабирование требует остановки экземпляров приложения. Это, конечно, не относится к приложениям, масштаб которых расширяется. Но с любым расширением приходит и сокращение. И для этого придется останавливать активные копии приложения. Если же взять вертикальное масштабирование, то перезапуск требуется для изменения объема выделенных ресурсов. В любом случае, большое значение будет иметь способность вашего приложения безопасно завершать свою работу. Эта тема подробно рассматривается в *главе 14*.

Итак, мы перечислили архитектурные аспекты, на которые следует обращать внимание. Теперь давайте подробно обсудим автомасштабирование приложений в кластерах Kubernetes.

Автомасштабирование приложений

Данный раздел посвящен автомасштабированию приложения. Это включает в себя мониторинг определенных метрик и изменение емкости приложений без вмешательства со стороны человека. Такой подход можно было бы охарактеризовать как "настроил и забыл", но не стоит к нему так относиться, особенно на начальных этапах. Даже после нагрузочного тестирования своих конфигураций автомасштабирования вам нужно убедиться в том, что ваша система ведет себя в реальном окружении так, как нужно. Нагрузочные тесты не всегда точно воспроизводят условия эксплуатации. Поэтому после окончательного развертывания приложения необходимо подтвердить, что оно масштабируется в подходящие моменты и удовлетворяет вашим требованиям к эффективности и удобству использования. Настоятельно рекомендуем установить оповещения, чтобы вы были в курсе важных событий масштабирования и при необходимости могли внести коррективы.

Бóльшая часть этого раздела посвящена компонентам HorizontalPodAutoscaler и VerticalPodAutoscaler. Это самые популярные средства автомасштабирования приложений в Kubernetes. Мы также обсудим метрики, с помощью которых ваши приложения инициируют события масштабирования, и объясним, в каких случаях следует подумать о создании для этой цели пользовательских метрик. Мы также уделим внимание компоненту cluster-proportional-autoscaler и покажем, в каких ситуациях его применение оправданно. В конце будет затронута тема нестандартных методов, выходящих за рамки вышеперечисленных инструментов.

Horizontal Pod Autoscaler

Horizontal Pod Autoscaler (HPA) — это самое распространенное средство автомасштабирования для платформ на основе Kubernetes. Его поддержка встроена в Kubernetes в виде одноименного ресурса и контроллера, входящего в состав kube-controller-manager. Если вы используете потребление ресурсов процессора или памяти в качестве метрик для автомасштабирования своих приложений, внедрение HPA не составит труда.

В данном случае, чтобы у HPA был доступ к метрикам Pod'ов, понадобится Kubernetes Metrics Server (**https://oreil.ly/S0vbj**). Это сервер, который извлекает из kubelets информацию об использовании процессора и памяти контейнерами кластера и делает ее доступной посредством API-интерфейса метрик в ресурсах PodMetrics. Metrics Server задействует слой агрегации в API-интерфейсе Kubernetes (**https://oreil.ly/eXDcl**). Запросы ресурсов в группе API и версия `metrics.k8s.io/v1beta1` будут направляться этому серверу.

На рис. 13.1 показано, как данные компоненты выполняют эту функцию. Сервер Metrics Server собирает метрики потребления ресурсов контейнерами платформы. Он получает эти данные от агента kubelet, запущенного на каждом узле кластера, и делает их доступными для клиентов, которым они нужны. По умолчанию контроллер HPA каждые 15 секунд обращается к API-серверу Kubernetes за информацией об использовании ресурсов. API-сервер передает эти запросы серверу метрик, который выдает соответствующие данные. Контроллер HPA следит за ресурсами типа `HorizontalPodAutoscaler` и на основе заданной в них конфигурации определяет, является ли адекватным число реплик приложения. В листинге 13.1 продемонстрировано, как это происходит. Приложение чаще всего имеет вид ресурса `Deployment`, когда контроллер HPA решает, что число реплик нужно отрегулировать, он обновляет соответствующий ресурс `Deployment` с помощью API-сервера. В ответ контроллер развертываний обновляет `ReplicaSet`, что приводит к изменению количества Pod'ов.

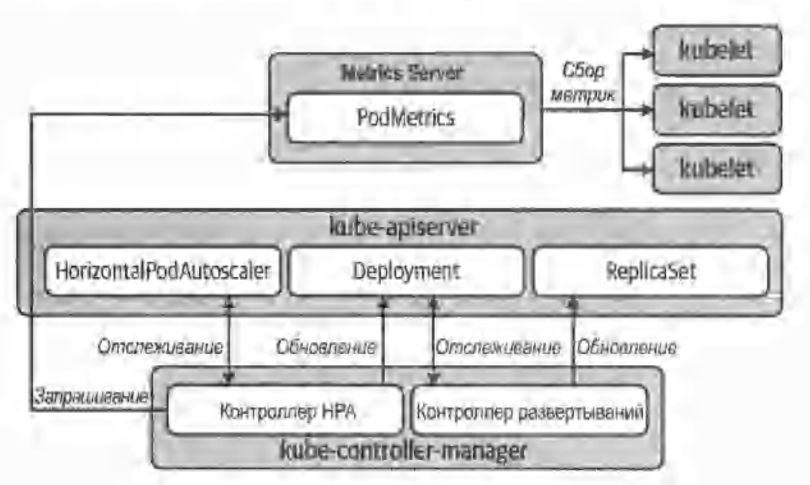


Рис. 13.1. Горизонтальное автомасштабирование Pod'ов

Как видно в следующем примере, желаемое состояние HPA объявлено в ресурсе `HorizontalPodAutoscaler`. Поле `targetCPUUtilizationPercentage` служит для определения числа реплик для того или иного приложения.

Листинг 13.1. Пример манифестов `Deployment` и `HorizontalPodAutoscaler`

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: sample
spec:
  selector:
    matchLabels:
      app: sample
  template:
    metadata:
      labels:
        app: sample
  spec:
    containers:
    - name: sample
      image: sample-image:1.0
      resources:
        requests:
          cpu: "100m" ❶
---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: sample
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sample
  minReplicas: 1 ❷
  maxReplicas: 3 ❸
  targetCPUUtilizationPercentage: 75 ❹
```

❶ Значение `resources.requests` должно быть указано для используемой метрики.

❷ Число реплик никогда не опустится ниже этого значения.

❸ Число реплик никогда не превысит это значение.

❹ Желаемая загруженность процессора. Число реплик увеличивается, если реальная нагрузка существенно превышает это значение, и уменьшается, если она слишком низкая.



Если вам нужно учитывать сразу несколько метрик (например, процессор и память), чтобы инициировать события масштабирования, можете воспользоваться API-интерфейсом `autoscaling/v2beta2`. В этом случае контроллер HPA будет определять подходящее число реплик с учетом каждой отдельной метрики и затем применять наибольшее значение.

Мы рассмотрели самый распространенный и доступный метод автомасштабирования, который имеет широкую область применения и относительно низкую сложность.

Однако вы должны понимать его ограничения:

- *Не все приложения могут масштабироваться горизонтально.* Для приложения, которое неспособно разделять нагрузку между несколькими экземплярами, горизонтальное масштабирование не имеет смысла. Это относится к некоторым приложениям, хранящим свое состояние, и приложениям с выбором лидера. В этих случаях стоит подумать о вертикальном автомасштабировании.
- *Масштабирование будет ограничено размером кластера.* Когда приложение расширяется, емкость рабочих узлов кластера может исчерпаться. Чтобы избежать этой проблемы, можно заранее выделять достаточное количество ресурсов (отправляя администраторам платформы оповещения, чтобы те увеличивали емкость вручную) или использовать автомасштабирование кластера, которое обсуждается в другом разделе этой главы.
- *Потребление процессора и памяти может быть неподходящей метрикой для принятия решений о масштабировании.* Если ваше приложение предоставляет собственную метрику, которая лучше сигнализирует о необходимости масштабирования, вы можете ее использовать. Данный подход будет рассмотрен далее в этой главе.



Не следует основывать автомасштабирование приложений на метрике, которая не всегда изменяется пропорционально нагрузке на приложение. Потребление ресурсов процессора — самая распространенная метрика для автомасштабирования. Но, если с повышением нагрузки на определенное приложение количество потребляемого им процессорного времени не меняется существенно, а вместо этого происходит прямо пропорциональное увеличение занятой им памяти, откажитесь от этой метрики.

Менее очевидный пример — приложение, которому нужны дополнительные вычислительные ресурсы на этапе запуска. В обычном режиме потребление ресурсов процессора может служить вполне полезным индикатором для автомасштабирования. Однако во время запуска повышенная нагрузка на процессор может быть интерпретирована контроллером HPA как повод для события масштабирования, хотя это повышение не связано с трафиком. Эту проблему можно минимизировать с помощью флагов kube-controller-manager таких, как `--horizontal-pod-autoscaler-cpu-initialization-period` (позволяет получить "отсрочку" на время запуска) или `--horizontal-pod-autoscaler-sync-period` (позволяет сделать проверки необходимости в масштабировании менее частыми). Но имейте в виду, что эти флаги предназначены для kube-controller-manager. Они повлияют на все контроллеры HPA в кластере и, следовательно, на приложения, которые не демонстрируют активного потребления ресурсов процессора во время запуска. В итоге вы можете ухудшить отзывчивость HPA в масштабе всего кластера. Если вы видите, что ваша команда ищет обходные пути, чтобы сделать потребление процессорного времени индикатором для нужд автомасштабирования, попробуйте найти более уместную пользовательскую метрику. Возможно, более подходящим показателем будет число полученных HTTP-запросов.

На этом мы заканчиваем обсуждение компонента HorizontalPodAutoscaler. Дальше будет рассмотрен другой вид автомасштабирования, доступный в Kubernetes: VerticalPodAutoscaler.

Vertical Pod Autoscaler

По причинам, рассмотренным ранее в разделе "Виды масштабирования", вертикальное масштабирование приложений требуется не так часто. Более того, в Kubernetes его сложнее автоматизировать. Если HPA входит в число основных компонентов Kubernetes, то VPA нужно реализовать самостоятельно путем развертывания трех отдельных контроллеров и Metrics Server в придачу. В связи с этим Vertical Pod Autoscaler (VPA; **https://oreil.ly/TxeiY**) применяется реже в сравнении с HPA.

VPA состоит из трех отдельных компонентов:

- *Рекомендатель* — определяет оптимальные значения для запросов процессора и/или памяти контейнеров заданного Pod'а с использованием ресурса PodMetrics.
- *Подключаемый модуль допуска* — изменяет запросы и лимиты на ресурсы для новых Pod'ов в момент их создания на основе информации от рекомендателя.
- *Средство обновления* — выгружает Pod'ы, чтобы модуль доступа мог применить к ним обновленные значения.

На рис. 13.2 проиллюстрировано взаимодействие компонентов с VPA.

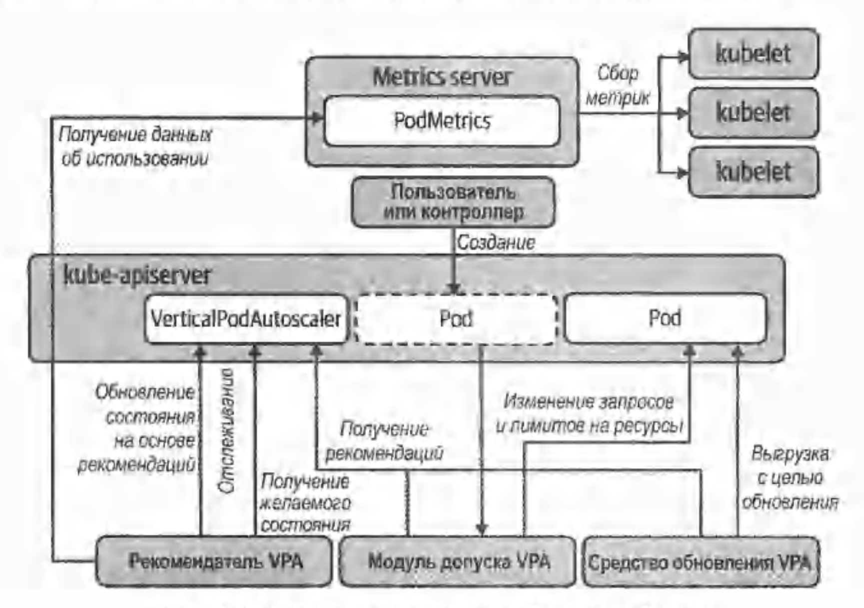


Рис. 13.2. Вертикальное автомасштабирование Pod'ов

Желаемое состояние VPA объявляется в ресурсе `VerticalPodAutoscaler`, как продемонстрировано в листинге 13.2.

Листинг 13.2. Ресурс `Pod` и ресурс `VerticalPodAutoscaler`, который конфигурирует вертикальное автомасштабирование

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: sample
spec:
  containers:
  - name: sample
    image: sample-image:1.0
    resources: ❶
      requests:
        cpu: 100m
        memory: 50Mi
      limits:
        cpu: 100m
        memory: 50Mi
---
apiVersion: "autoscaling.k8s.io/v1beta2"
kind: VerticalPodAutoscaler
metadata:
  name: sample
spec:
  targetRef:
    apiVersion: "v1"
    kind: Pod
    name: sample
  resourcePolicy:
    containerPolicies:
      - containerName: '*' ❷
        minAllowed: ❸
          cpu: 100m
          memory: 50Mi
        maxAllowed: ❹
          cpu: 1
          memory: 500Mi
        controlledResources: ["cpu", "memory"] ❺
  updatePolicy:
    updateMode: Recreate ❻
```

❶ VPA будет поддерживать соотношение запросов и лимитов при обновлении значений. В этом примере с гарантированным QoS любое изменение запросов будет вызывать идентичное изменение лимитов.

❷ Эта политика масштабирования будет применяться к каждому контейнеру (в данном примере только к одному).

❸ Запросы ресурсов не опустятся ниже этих значений.

❹ Запросы ресурсов не превысят эти значения.

❺ Указывает ресурсы, которые автоматически масштабируются.

❻ У параметра `updateMode` есть три возможных значения. Режим `Recreate` активирует автомасштабирование. Режим `Initial` применяет управление допуском к набору

значений ресурсов при их создании, но никогда не выгружает никакие Pod'ы. Режим `Off` предлагает значения ресурсов в качестве рекомендации, но никогда не изменяет их автоматически.

В реальных условиях VPA в полноценном режиме `Recreate` встречается нам крайне редко. Но даже в режиме `Off` автомасштабирование VPA может быть полезным. Перед развертыванием приложений в среде эксплуатации рекомендуется провести их всестороннее нагрузочное тестирование и профилирование, но в реальности это происходит не всегда. В корпоративных окружениях с крайними сроками приложения нередко развертываются для использования без надлежащего анализа потребления ресурсов. Это зачастую приводит к тому, что ресурсы на всякий случай запрашиваются в избытке, что может привести к неоптимальной загрузке инфраструктуры. В таких случаях с помощью VPA можно получать рекомендуемые значения, которые затем взвешиваются и вручную обновляются инженерами в условиях реальных нагрузок. Это даст им уверенность в том, что приложения не будут выгружены в периоды пикового потребления, что особенно важно для приложений, которые еще не умеют безопасно завершать свою работу. Это также позволяет сэкономить некоторые усилия, направленные на анализ потребления ресурсов и определение оптимальных значений. В таких случаях VPA выступает не автоматическим средством масштабирования, а, скорее, вспомогательным инструментом для регулирования ресурсов.

Чтобы получать рекомендации от VPA в режиме `Off`, выполните команду `kubectl describe vpa <название vpa>`. Вы получите примерно такой вывод, как показано в листинге 13.3 в разделе `Status`.

Листинг 13.3. Рекомендации от VerticalPodAutoscaler

```
Recommendation:
  Container Recommendations:
    Container Name:  coredns
    Lower Bound:
      Cpu:     25m
      Memory:  262144k
    Target:
      Cpu:     25m
      Memory:  262144k
    Uncapped Target:
      Cpu:     25m
      Memory:  262144k
    Upper Bound:
      Cpu:     427m
      Memory:  916943343
```

Вы получите рекомендации для каждого контейнера. Используйте величины из раздела `Target` в качестве исходного значения для запросов процессора и памяти.

Автомасштабирование с помощью пользовательских метрик

Если потребление ресурсов процессора и памяти не подходит для принятия решений о масштабировании определенного приложения, то в качестве альтернативы можно обратиться к пользовательским метрикам. Мы по-прежнему можем применять такие инструменты как HPA, просто нужно изменить источник метрик, которые инициируют автомасштабирование. Для начала следует сделать так, чтобы ваше приложение предоставляло подходящие пользовательские метрики. В *главе 14* обсуждается то, как это можно сделать.

Дальше вам нужно сделать эти пользовательские метрики доступными для средства автомасштабирования. Для этого понадобится отдельный сервер метрик, который заменит собой сервер Kubernetes Metrics Server, рассмотренный ранее. Некоторые поставщики, такие как Datadog, предоставляют для такой ситуации специальные системы. Но это также можно сделать с помощью Prometheus, при условии, что у вас уже есть сервер Prometheus, который собирает и сохраняет пользовательские метрики приложения (см. *главу 10*). В этом случае для предоставления пользовательских метрик можно применить Prometheus Adapter (**https://oreil.ly/vDgk3**).

Prometheus Adapter будет извлекать пользовательские метрики из HTTP API Prometheus и делать их доступными посредством API-интерфейса Kubernetes. Prometheus Adapter, как и Metrics Server, с помощью средств агрегирования Kubernetes направляет запросы, предназначенные для API-интерфейса метрик, к Prometheus Adapter. На самом деле, помимо API-интерфейса пользовательских метрик, Prometheus Adapter реализует интерфейс для работы с ресурсами, позволяя полностью заменить собой Metrics Server. Кроме того, он реализует интерфейс для внешних метрик, который дает возможность масштабировать приложение с учетом показателей, собранных за пределами кластера.

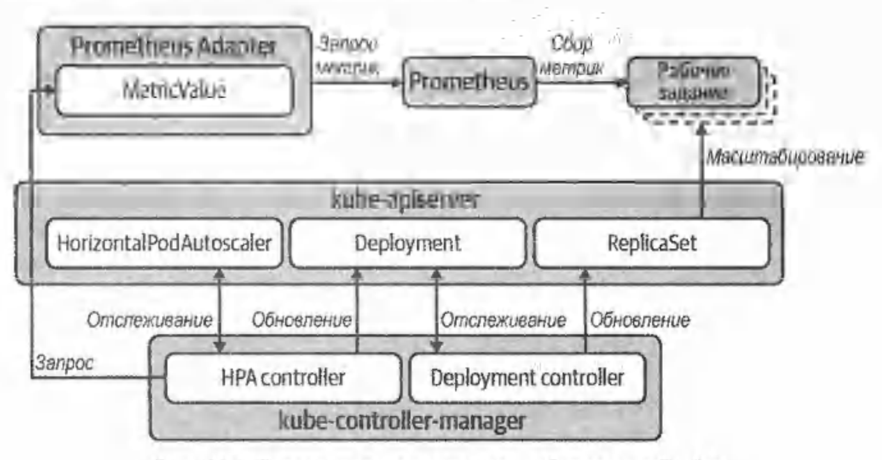


Рис. 13.3. Горизонтальное автомасштабирование Pod'ов с помощью пользовательских ресурсов

Если у вашего приложения есть пользовательские метрики для горизонтального автомасштабирования, Prometheus их соберет и передаст их системе Prometheus Adapter, которая сделает их доступными через API-сервер Kubernetes. HPA будет запрашивать эти метрики и масштабировать ваше приложение соответствующим образом, как показано на рис. 13.3.

Такое применение пользовательских метрик создает дополнительные сложности, но, если ваши приложения уже предоставляют полезные метрики, которые отслеживаются с помощью Prometheus, то переход с Metrics Server на Prometheus Adapter не потребует огромных изменений. А дополнительные возможности для автомасштабирования, которые открывает этот подход, делают его заслуживающим внимания.

cluster-proportional-autoscaler

Компонент cluster-proportional-autoscaler (CPA; **https://oreil.ly/2ATBG**) — это горизонтальное средство автомасштабирования приложений, которое принимает свои решения с учетом общего числа (или подмножества) узлов в кластере. Поэтому, в отличие от HPA, оно не полагается ни на какие API-интерфейсы метрик. В связи с этим оно не зависит от Metrics Server или Prometheus Adapter. Кроме того, cluster-proportional-autoscaler настраивается не с помощью ресурса Kubernetes, а с использованием флагов для конкретных приложений и ресурса `ConfigMap` для параметров масштабирования. На рис. 13.4 проиллюстрирован намного более простой принцип работы CPA.

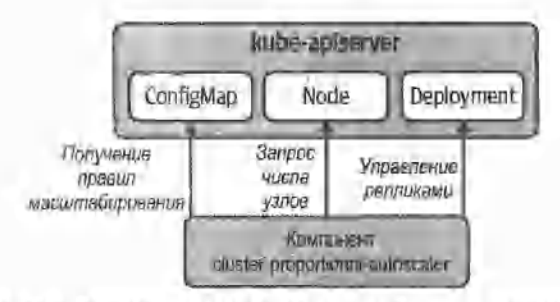


Рис. 13.4. Пропорциональное автомасштабирование кластера

Область применения CPA более узкая. К приложениям, которым нужно масштабироваться пропорционально кластеру, обычно относят только сервисы платформы. Рассматривая возможность использования CPA, подумайте о том, будет ли система HPA лучшим решением, особенно если вы уже применяете ее для других приложений. Если в вашем кластере уже есть HPA, значит, у вас уже развернуты Metrics Server или Prometheus Adapter, которые реализуют необходимые API-интерфейсы метрик. Поэтому развертывание еще одного средства автомасштабирования может быть не самым оптимальным выбором, учитывая накладные расходы на его обслуживание. Если же HPA *не* используется в вашем кластере, а CPA предоставляет нужные вам возможности, такой вариант может стать более привлекательным благодаря своему простому принципу работы.

CPA поддерживает два метода масштабирования:

- линейное масштабирование приложений пропорционально числу узлов или процессорных ядер в кластере;
- определение соотношения между узлами/ядрами и репликами на основе ступенчатой функции.

В нашей работе встречаются примеры успешного применения CPA для таких сервисов, как DNS, когда кластерам позволено масштабировать сотни рабочих узлов. В подобных случаях трафик и востребованность сервисов на уровне 5 и 300 узлов будут совершенно разными, поэтому данный подход может быть довольно полезным.

Создание собственных средств автомасштабирования

В рамках темы автомасштабирования приложений мы рассмотрели определенные инструменты, предлагаемые сообществом Kubernetes: HPA, VPA и CPA, а также Metrics Server и Prometheus Adapter. Но вам необязательно ограничиваться этими инструментами. К той же категории можно отнести любой автоматизированный метод, реализующий нужное вам поведение. Например, если известны дата и время пиковой нагрузки на ваше приложение, то можно воспользоваться таким простым средством, как объект `CronJob` из состава Kubernetes, который будет обновлять число реплик для соответствующего Развертывания. На самом деле, если у вас есть возможность использовать простой и понятный метод вроде этого, ему следует отдавать предпочтение. Чем проще система, тем меньше вероятность получения неожиданных результатов.

На этом мы завершаем обсуждение методов автомасштабирования приложений. Мы рассмотрели несколько подходов к этой проблеме, основанных на решениях, встроенных в Kubernetes, разрабатываемых сообществом и написанных самостоятельно. Дальше речь пойдет об автомасштабировании того слоя, в котором размещаются приложения: самого кластера Kubernetes.

Автомасштабирование кластера

Kubernetes Cluster Autoscaler (CA) (**https://oreil.ly/Q5Xdp**) предоставляет автоматизированное средство горизонтального масштабирования рабочих узлов кластера, решает одну из проблем, присущих механизмам HPA, и существенно облегчает управление денежными расходами на инфраструктуру Kubernetes и ее емкостью.

По мере внедрения платформы, основанной на Kubernetes, вам придется управлять емкостью кластера с учетом добавления новых арендаторов. Данный процесс может быть ручным и рутинным. В его основе также могут лежать оповещения, правила генерации которых зависят от показателей потребления ресурсов и уведомляют вас о необходимости добавления или удаления рабочих узлов. Или же вы можете полностью автоматизировать этот процесс, чтобы средство CA управляло масштабированием кластера в ответ на добавление или удаление арендаторов.

Более того, в условиях существенных колебаний в потреблении ресурсов, вызванных автомасштабированием приложений, применение CA становится еще более оправданным. С повышением нагрузки на приложения под управлением HPA увеличивается и число их реплик. Если у вашего кластера закончатся вычислительные ресурсы, некоторые Pod'ы не будут развернуты и останутся в состоянии `Pending`. CA распознает эту конкретную ситуацию, вычисляет число узлов, необходимых для преодоления дефицита ресурсов, и добавляет новые узлы в кластер. На рис. 13.5 продемонстрировано расширение кластера с целью удовлетворить потребности горизонтально масштабируемого приложения.

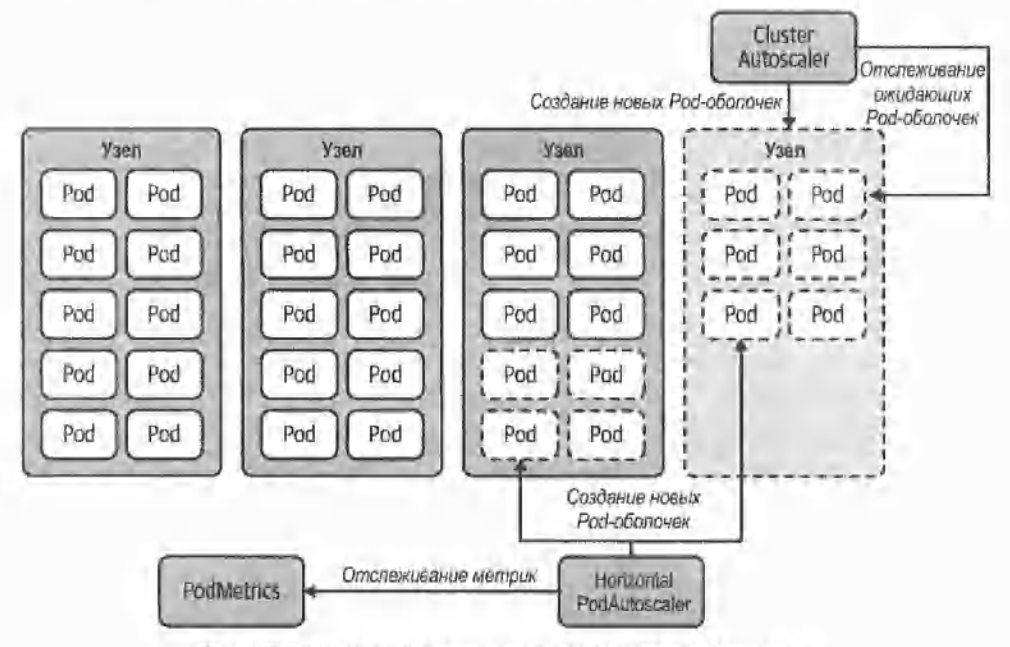


Рис. 13.5. Cluster Autoscaler увеличивает число узлов в ответ на увеличение числа экземпляров Pod

С другой стороны, когда нагрузка падает и HPA сокращает число Pod'ов приложения, CA ищет узлы, которые недостаточно использовались на протяжении длительного времени. Если Pod'ы на таких узлах можно перенести в другое место, CA отзовет эти узлы, чтобы сократить масштаб кластера.

Используя этот динамический механизм управления рабочими узлами, вы должны помнить о том, что он неизбежно изменит распределение Pod'ов в вашем кластере. Планировщик Kubernetes, как правило, распределяет Pod'ы равномерно при их создании и в дальнейшем не пересматривает решения о месте их размещения, пока не придет время их выгружать. Поэтому, когда определенное приложение расширяется и затем сокращается за счет горизонтального масштабирования, вы можете получить неравномерное распределение Pod'ов по вашим рабочим узлам. В некоторых случаях множество копий Развертывания может оказаться всего на нескольких узлах. Если это ставит под угрозу отказоустойчивость узлов с приложениями, мож-

но задействовать Kubernetes descheduler (**https://github.com/kubernetes-sigs/descheduler**), чтобы выгрузить Pod'ы в соответствии с разными политиками, после чего они будут развернуты заново. В результате распределение Pod'ов по узлам будет сбалансировано. Нам редко встречаются ситуации, в которых так действительно нужно делать, но это один из возможных вариантов.

Как можно себе представить, для автомасштабирования кластера необходимо учитывать вопросы, связанные с управлением инфраструктурой. Прежде всего, вам нужно пользоваться услугами одного из поддерживаемых облачных провайдеров, список которых задокументирован в репозитории проекта. В дополнение к этому вы должны выдать средству CA права, необходимые для создания и удаления узлов.

Эти аспекты администрирования инфраструктуры в некоторой степени зависят от того, используете ли вы CA в сочетании с проектом Cluster API (**https://github.com/kubernetes-sigs/cluster-api**). У Cluster API есть свои операторы для управления инфраструктурой. В этом случае для добавления или удаления рабочих узлов CA не обращается непосредственно к облачному провайдеру, а делегирует эти операции Cluster API. CA просто обновляет число реплик в ресурсе `MachineDeployment`, который затем согласовывается контроллерами Cluster API. Благодаря этому нам необязательно пользоваться облачным провайдером, совместимым с CA (хотя вам *нужно* убедиться в том, что у Cluster API есть подходящий провайдер). Проблема с правами доступа тоже перекладывается на компоненты Cluster API. Такая модель более удачная во многих отношениях. С другой стороны, Cluster API обычно реализуется с помощью управляющих кластеров, что создает дополнительные зависимости для CA, которые необходимо принимать во внимание. Эта тема подробно обсуждается в разделе "Управляющие кластеры" *главы 2*.

То, как CA выполняет масштабирование, довольно гибко настраивается. Для этого предусмотрены флаги, задокументированные в разделе с часто задаваемыми вопросами на GitHub этого проекта (**https://oreil.ly/DzQ0J**). В листинге 13.4 приведен пример манифеста Развертывания CA для AWS, который демонстрирует, как устанавливаются некоторые распространенные флаги.

Листинг 13.4. Манифест Развертывания CA для группы автомасштабирования Amazon Web Services

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aws-cluster-autoscaler
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: "aws-cluster-autoscaler"
  template:
    metadata:
```

```
  labels:
    app.kubernetes.io/name: "aws-cluster-autoscaler"
spec:
  containers:
    - name: aws-cluster-autoscaler
      image: "us.gcr.io/k8s-artifacts-prod/autoscaling/cluster-autoscaler:v1.18"
      command:
        - ./cluster-autoscaler
        - --cloud-provider=aws ❶
        - --namespace=kube-system
        - --nodes=1:10:worker-auto-scaling-group ❷
        - --logtostderr=true
        - --stderrthreshold=info
        - --v=4
      env:
        - name: AWS_REGION
          value: "us-east-2"
      livenessProbe:
        httpGet:
          path: /health-check
          port: 8085
      ports:
        - containerPort: 8085
```

❶ Определение поддерживаемого провайдера; в данном случае AWS.

❷ Этот флаг настраивает CA для обновления группы автомасштабирования AWS под названием `worker-auto-scaling-group`. Это позволяет CA изменять число серверов в данной группе в пределах от 1 до 10.

Автомасштабирование кластера может быть чрезвычайно полезным. Оно позволяет воспользоваться одним из важных преимуществ облачно-ориентированной инфраструктуры. Но вместе с тем оно существенно усложняет вашу систему. Не забудьте провести нагрузочное тестирование и попытайтесь досконально разобраться в том, как ведет себя ваша система, прежде чем позволять CA самостоятельно управлять масштабированием критически важных платформ в реальных условиях. Вы должны четко понимать, какая максимальная нагрузка будет оказываться на ваш кластер. Если ваша платформа обслуживает существенное число приложений, и вы позволяете своему кластеру расширяться до сотен узлов, вам нужно определить, какие его компоненты будут масштабироваться, прежде чем они начнут отрицательно влиять на производительность. Подробней о том, как выбрать размер кластера, можно почитать в *главе 2*.

Еще один важный фактор автомасштабирования кластера — скорость, с которой он будет масштабироваться при возникновении такой необходимости. Здесь может помочь выделение резервных ресурсов.

Выделение резервных ресурсов для кластера

Необходимо помнить, что Cluster Autoscaler реагирует на Pod'ы в состоянии `Pending`, которые не удается развернуть из-за нехватки ресурсов в кластере. Поэтому в момент, когда CA начинает увеличивать число узлов, ваш кластер уже заполнен до отказа. Если масштабирование приложений выполнить неправильно, то в момент, когда новые узлы станут доступными для развертывания, у вас может возникнуть дефицит ресурсов. Решить эту проблему поможет средство cluster-overprovisioner (**https://oreil.ly/vXij5**).

Для начала необходимо понять, сколько времени нужно новым узлам для того, чтобы загрузиться, присоединиться к кластеру и подготовиться к приему приложений. Определившись с этим, вы можете выбрать лучшее решение для вашей ситуации:

- Укажите достаточно низкий целевой уровень загруженности в своих ресурсах HPA, чтобы ваши приложения хорошо отмасштабировались, прежде чем приложение достигнет своей максимальной емкости. Это послужит буфером, который даст время на выделение узлов и избавит от необходимости выделения резервных ресурсов в кластере. Но, если вам нужно реагировать на особенно резкие скачки нагрузки, уровень целевой загруженности, вероятно, придется сделать слишком низким, что не позволит уберечься от нехватки емкости. В результате возникнет ситуация, когда у вас постоянно выделяются лишние ресурсы на случай возникновения редких событий.
- Еще одно решение заключается в выделении резервных ресурсов. Мы держим в резерве узлы, которые будут служить буфером для масштабируемых приложений. Это избавит от необходимости делать целевую загруженность для HPA намеренно заниженной с расчетом на скачки нагрузки.

Резервные ресурсы для кластера выделяют путем развертывания Pod'ов, которые:

- запрашивают достаточное количество ресурсов, чтобы их можно было практически целиком зарезервировать для узла;
- сами не потребляют никаких реальных ресурсов;
- учитывают класс приоритета, благодаря которому они выгружаются сразу, как только их ресурсы понадобятся какому-то другому Pod'у.

Указав запросы ресурсов, которые зарезервируют для Pod'а целый узел, вы сможете регулировать число резервных узлов, изменяя значение реплик в соответствующем ресурсе `Deployment`. Для выделения резервных ресурсов на случай какого-то конкретного события или рекламной акции достаточно увеличить число реплик.

На рис. 13.6 показано, как это выглядит. Здесь изображена всего одна копия Pod'а, но число таких копий ограничено лишь вашей потребностью в предоставлении адекватного буфера для событий масштабирования.

Теперь зарезервированный узел в любой момент может быть предоставлен другому Pod'у, которому он нужен. Для этого создается класс приоритета со значением `value: -1` и применяется к объекту `Deployment`, который занимается выделением резервных ресурсов. В результате все остальные приложения автоматически получат

более высокий приоритет. Если Pod'у другого приложения потребуются ресурсы, то зарезервированный узел будет немедленно освобожден и предоставлен масштабируемому приложению. Pod, занимавший резервный узел, перейдет в состояние `Pending`, что заставит Cluster Autoscaler выделить новый узел, который будет находиться в резерве. Это проиллюстрировано на рис. 13.7.

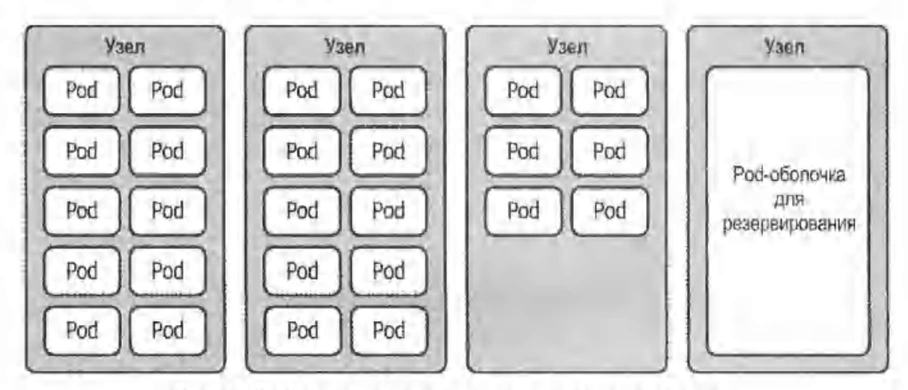


Рис. 13.6 Выделение резервных ресурсов для кластера

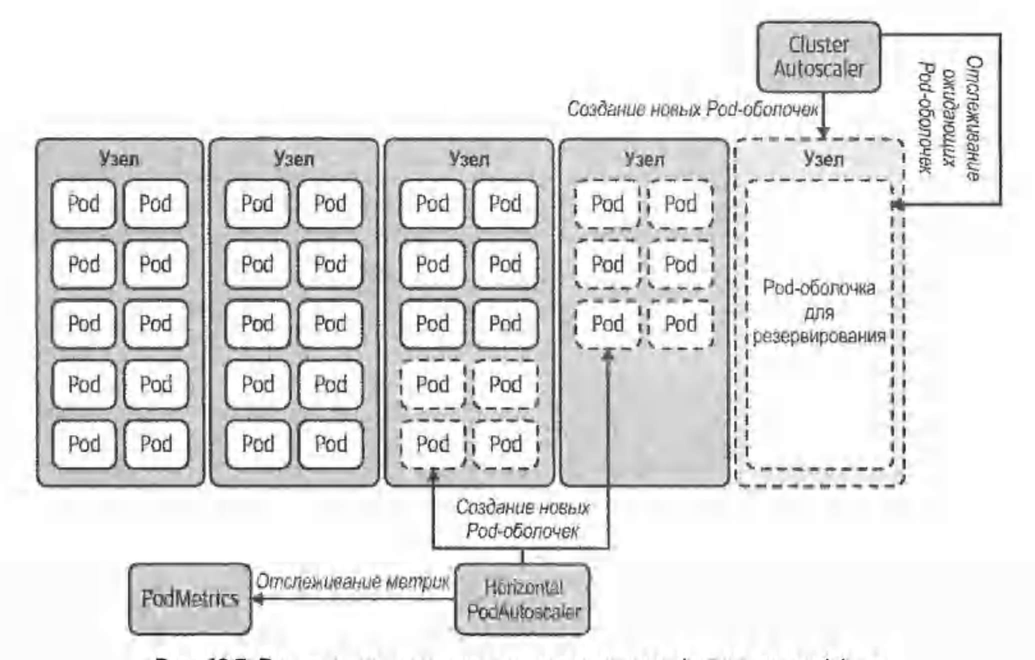


Рис. 13.7 Расширение кластера с использованием cluster-overprovisioner

Компоненты Cluster Autoscaler и cluster-overprovisioner — эффективные механизмы горизонтального масштабирования кластеров Kubernetes, которые отлично сочетаются с горизонтально масштабируемыми приложениями. Мы не стали рассматривать вертикальное масштабирование кластеров, так как нам не удалось найти ему вариант применения, в котором нельзя было бы обойтись горизонтальным масштабированием.

Резюме

Если у вас есть приложения, у которых могут существенно изменяться требования к емкости, по возможности отдавайте предпочтение горизонтальному масштабированию. Разрабатывайте автомасштабируемые приложения так, чтобы их можно было регулярно останавливать и запускать, и чтобы они предоставляли пользовательские метрики, если показатели процессора или памяти не подходят для управления масштабированием. Протестируйте свои средства автомасштабирования и убедитесь в том, что они ведут себя предсказуемо. В результате ваша система станет более эффективной и удобной в использовании. Если при масштабировании вашим приложениям не хватает ресурсов, подумайте о том, чтобы добавить поддержку автомасштабирования в сам кластер. И если у вас случаются особенно резкие скачки нагрузки, попробуйте подготовить резервные узлы с помощью средства cluster-overprovisioner.

ГЛАВА 14

Эффективная эксплуатация приложений

Платформа Kubernetes позволяет выполнять и администрировать довольно широкий спектр приложений. Если не считать ограничений, накладываемых операционной системой и архитектурой процессора, в Kubernetes может работать фактически все что угодно: крупные монолитные системы, распределенные микросервисы, пакетные рабочие и т. д. Единственное требование, которое Kubernetes предъявляет к приложениям, состоит в том, что они должны распространяться в виде образов контейнеров. Тем не менее, вы можете предпринять определенные шаги для того, чтобы сделать работу своих приложений в Kubernetes более эффективной.

В этой главе основной темой нашего обсуждения будет не платформа, а приложения. Если вы являетесь членом команды, которая обслуживает платформу, то можете подумать, что эта глава предназначена лишь для разработчиков, но она будет полезна и вам. В ходе своей работы над платформой вам почти наверняка придется создавать приложения для предоставления нестандартных сервисов. Но, даже если это не так, обсуждаемые здесь темы помогут вам лучше взаимодействовать с разработчиками, которые пользуются вашей платформой. Возможно, вам даже будет чему научить тех программистов, которые не знакомы с платформами, основанными на контейнерах.

В этой главе рассматриваются различные аспекты размещения приложений в Kubernetes:

- Развертывание приложений на платформе и механизмы для управления манифестами развертывания такие, как шаблонизация и упаковка.
- Методы настройки приложений, в том числе с использованием API-интерфейсов Kubernetes (ConfigMap/Secret) и на основе интеграции с внешними системами для управления конфигурацией и конфиденциальными данными.
- Механизмы Kubernetes, улучшающие доступность ваших приложений, такие, как хуки, срабатывающие перед остановкой контейнеров, безопасное принудительное завершение и ограничения планирования.
- Проверки состояния — функция Kubernetes, позволяющая передавать платформе информацию о работоспособности приложения.
- Запросы и лимиты на ресурсы, необходимые для корректной работы приложений на платформе.
- Журналирование, система метрик и трассировка в качестве механизмов для эффективной отладки, диагностики и администрирования.

Развертывание приложений в Kubernetes

Упаковав свое приложение в контейнер и сделав его доступным в реестре контейнерных образов, вы готовы к тому, чтобы развернуть его в Kubernetes. В большинстве случае развертывание приложения подразумевает написание манифестов в формате YAML с описанием ресурсов Kubernetes, необходимых для его работы, таких как Deployment, Service, ConfigMap, CRD и т. д. Затем эти манифесты передаются API-серверу, а Kubernetes заботится обо всем остальном. Обычные YAML-манифесты годятся для начала, но их практичность может быстро сойти на нет, особенно при развертывании одного и того же приложения в разных кластерах или окружениях. У вас почти наверняка возникнут следующие вопросы:

- Как предоставлять разные учетные данные при работе в среде финального тестирования и в реальных условиях?
- Как использовать разные реестры образов при развертывании в различных центрах обработки данных?
- Как установить разное число реплик в отладочном и реальном окружении?
- Как сделать так, чтобы в разных манифестах совпадали номера всех портов?

Данный список можно продолжать. Каждую из этих задач можно решить за счет наличия нескольких наборов манифестов, однако переключаться между ними будет довольно непросто. Далее мы обсудим меры, которые можно принять, чтобы упростить управление манифестами. Речь в основном пойдет о шаблонизации манифестов и упаковке приложений для Kubernetes. Однако мы не станем рассматривать многочисленные инструменты, разрабатываемые сообществом. Как показывает наш опыт, большинство команд разработчиков впадают в ступор при выборе из множества разных вариантов. Мы советуем выбрать *что-нибудь*, неважно что, и побыстрее переходить к задачам более общего характера.

Шаблонизация манифестов развертывания

Шаблонизация подразумевает добавление подставляемых параметров в манифесты развертывания. Вместо того чтобы прописывать значения прямо в манифестах, мы задействуем механизм, который позволяет внедрять их по мере необходимости. Например, шаблонизированный манифест дает возможность указывать разное число реплик. На этапе разработки вам может хватить одной реплики, а в реальном окружении вам нужно сразу пять (листинг 14.1).

Листинг 14.1

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
```

```
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
```

Упаковка приложений для Kubernetes

Создание автономных программных пакетов — это еще один механизм, который облегчает управление манифестами, и пригоден также для развертывания приложений. Системы упаковывания обычно основаны на шаблонизации, но предоставляют дополнительные функции, которые могут оказаться полезными, например, возможность загрузки пакетов в реестры, совместимые с OCI, хуки жизненного цикла и др.

Пакеты — замечательный механизм использования стороннего программного обеспечения и доставки собственного ПО третьим лицам. Если вы применяете Heml для установки ПО в кластере Kubernetes, вам уже известны преимущества этого подхода. Но, если вы не знакомы с Helm, следующий пример поможет вам получить представление о том, как выглядит процесс установки пакета:

```
$ helm repo add hashicorp https://helm.releases.hashicorp.com
"hashicorp" has been added to your repositories
$ helm install vault hashicorp/vault
```

Как видите, пакеты могут служить отличным средством развертывания и администрирования программного обеспечения в Kubernetes. С другой стороны, они могут плохо подходить для приложений, которым требуется расширенное управление жизненным циклом. Мы считаем, что для таких приложений целесообразнее операторы, подробно рассмотренные в *главе 2*. Несмотря на то, что основное внимание в данной главе уделяется сервисам платформы, аналогичные принципы применимы и к процессу создания операторов для сложных приложений.

Получение конфигурации и конфиденциальных данных

У приложений обычно есть конфигурация, которая определяет их поведение во время работы. В конфигурацию обычно входят уровни журналирования, сетевые имена зависимостей (например, DNS-запись для базы данных), время ожидания

и т. д. Некоторые из этих параметров могут содержать конфиденциальную информацию вроде паролей, которую мы обычно называем секретной. В этом разделе мы обсудим разные методы, с помощью которых можно конфигурировать приложения на платформе, основанной на Kubernetes. Сначала дадим краткий обзор API-интерфейсов ConfigMap и Secret, встроенных в Kubernetes. Затем мы исследуем их альтернативы, которые в основном интегрируются с внешними системами. В завершение мы поделимся рекомендациями относительно этих подходов, исходя из того, какие решения лучше всего работают на практике.

Прежде чем углубляться в эту тему, стоит упомянуть, что конфигурацию или конфиденциальные данные не следует включать в образ контейнера приложения. Жесткое связывание исполняемого файла и его конфигурации лишает смысла саму идею изменения параметров во время выполнения. Более того, в случае с конфиденциальными данными это ставит под угрозу безопасность, так как образ может быть доступен тем, у кого не должно быть доступа к таким данным. Вместо того чтобы добавлять конфигурацию в образ, ее нужно внедрять на этапе выполнения, используя возможности платформы.

Объекты *ConfigMap* и *Secret*

Объекты `ConfigMap` и `Secret` — это основные ресурсы API-интерфейса Kubernetes, которые позволяют настраивать приложения во время их работы. Как и любой другой ресурс Kubernetes, они создаются средствами API-сервера и обычно объявляются в формате YAML, как показано в следующем примере:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  debug: "false"
```

Давайте поговорим о том, как вы можете использовать эти ресурсы в своих приложениях.

Первый метод состоит в подключении объектов `ConfigMap` и `Secret` к файловой системе Pod'а. При создании спецификации Pod можно указать том, который ссылается на объекты `ConfigMap` и `Secret` по их названиям, и подключить их к определенным директориям. Например, в листинге 14.2 фрагмент кода определяет Pod, который подключает `ConfigMap` с именем `my-config` к контейнеру под названием `my-app`, делая его доступным по пути /etc/my-app/config.json.

Листинг 14.2

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
```

```
spec:
  containers:
  - image: my-app
    name: my-app:v0.1.0
    volumeMounts:
    - name: my-config
      mountPath: /etc/my-app/config.json
  volumes:
  - name: my-config
    configMap:
      name: my-config
```

Подключение томов является предпочтительным методом потребления объектов `ConfigMap` и `Secret`. Дело в том, что файлы в Pod'е динамически обновляются, что позволяет изменять конфигурацию приложений без их перезапуска или создания нового Pod'а. Тем не менее, эта возможность должна поддерживаться самим приложением. Приложение должно следить за конфигурационными файлами на диске и применять новую конфигурацию при их изменении. Существует множество библиотек и платформ, которые облегчают реализацию такого подхода. Но, если их использование невозможно, вы можете добавить sidecar контейнер, который будет отслеживать конфигурационные файлы и оповещать главный процесс (например, с помощью сигнала `SIGHUP`) о наличии новой конфигурации.

Объекты `ConfigMap` и `Secret` можно использовать с помощью переменных окружения. Если ваше приложение уже получает конфигурацию с помощью такого механизма, то данный подход будет естественным. Переменные окружения также могут пригодиться, если вам нужно предоставлять настройки в виде флагов командной строки. В следующем примере (листинг 14.3) Pod устанавливает переменную окружения `DEBUG`, используя объект `ConfigMap` с именем `my-config`, у которого есть ключ `debug` с соответствующим значением.

Листинг 14.3

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
    - name: my-app
      image: my-app:v0.1.0
      env:
        - name: DEBUG
          valueFrom:
            configMapKeyRef:
              name: my-config
              key: debug
```

Один из недостатков использования переменных окружения состоит в том, что изменения, вносимые в объекты `ConfigMap` и `Secret`, не отражаются на активном Pod'е, пока тот не перезапустится. В некоторых случаях это может быть приемлемо, но вы должны об этом помнить. Еще один недостаток, который в основном касается объектов `Secret`, — они во время запуска или сбоя приложения или платформы могут сбрасывать информацию об окружении в журнал. Это угроза безопасности, так как конфиденциальные данные могут случайно просочиться в журнальные записи.

Рассмотренные два метода потребления `ConfigMap` и `Secret` работают благодаря тому, что Kubernetes внедряет конфигурацию в приложения. Но есть и другой вариант: приложение может само обратиться к API-интерфейсу Kubernetes за своей конфигурацией. Вместо конфигурационных файлов и переменных окружения оно будет получать объекты `ConfigMap` и `Secret` непосредственно от API-сервера. Приложение также может следить за API-интерфейсом и реагировать на любые изменения конфигурации. В распоряжении разработчиков есть множество библиотек и SDK, позволяющих реализовать эти функции, они также могут воспользоваться платформами разработки с поддержкой этой возможности, такими как Spring Cloud Kubernetes.

Получение конфигурации через API-интерфейс Kubernetes может быть удобно, но мы считаем, что у этого подхода есть важные недостатки, которые необходимо учитывать. Во-первых, необходимость обращаться к API-серверу за конфигурацией жестко привязывает приложение к платформе Kubernetes. В результате возникают интересные вопросы. Например, что будет, если API-сервер выйдет из строя? Или будет ли ваше приложение недоступным, пока администраторы платформы обновляют API-сервер?

Во-вторых, чтобы достать свою конфигурацию из API-интерфейса, приложению нужны учетные данные и подходящие права доступа. Эти требования делают процесс развертывания более сложным, так как вам нужно предоставить своим приложениям служебную учетную запись и определить роли RBAC.

И последнее замечание: чем больше приложений получают конфигурацию посредством этого метода, тем больше нагрузка на API-сервер. В связи с тем, что API-сервер — один из ключевых компонентов плоскости управления, описанный подход к конфигурации приложений может плохо сказываться на масштабируемости кластера.

В целом, когда речь идет о потреблении объектов `ConfigMap` и `Secret`, мы предпочитаем использовать подключаемые тома и переменные окружения, вместо того чтобы обращаться к API-интерфейсу Kubernetes напрямую. Это позволяет делать приложения независимыми от платформы, на которой они выполняются.



Бывают ситуации, когда приложениям нужно получить сведения о самих себе. Возможно, им нужно узнать, в каком пространстве имен они выполняются, какие у них метки или лимиты на ресурсы. Kubernetes предоставляет API-интерфейс Downward, который позволяет внедрять метаданные Pod'ов так, чтобы приложениям не нужно было взаимодействовать с Kubernetes или знать о ней. Подобно объектам `ConfigMap` и `Secret`, эти метаданные можно делать доступными в виде переменных окружения или подключаемых томов.

В листинге 14.4 приведен пример использования API-интерфейса Downward. В данном случае Pod'у нужно знать, какой у нее лимит на память, эта информация доступна в виде переменной окружения с именем `MEM_LIMIT`.

Листинг 14.4

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
    - name: my-app
      image: my-app:0.1.0
      command: [ "my-app" ]
      env:
        - name: MEM_LIMIT
          valueFrom:
            resourceFieldRef:
              containerName: my-app
              resource: limits.memory
```

Получение конфигурации из внешних систем

Объекты `ConfigMap` и `Secret` могут быть удобным средством конфигурации приложений. Они встроены в API-интерфейс Kubernetes и доступны для свободного использования. С другой стороны, конфигурация и конфиденциальные данные — это проблема, с которой разработчики имели дело задолго до появления Kubernetes. И хотя Kubernetes предлагает механизмы для ее решения, вам ничто не мешает прибегнуть к внешним системам.

Одним из самых распространенных примеров внешней системы для управления конфигурацией или конфиденциальными данными, с которыми мы сталкивались, является HashiCorp Vault (**https://www.vaultproject.io**). Vault предлагает расширенные возможности управления конфиденциальной информацией, недоступные стандартным объектам `Secret`. Например, Vault поддерживает динамические конфиденциальные данные и их ротацию, временные токены и т. д. Если ваше приложение уже использует Vault, оно может и дальше это делать при работе в Kubernetes. В противном случае вам стоит подумать о внедрении этой системы в качестве более функциональной альтернативы объектам `Secret`. Мы уже подробно обсуждали различные аспекты управления конфиденциальными данными и интеграцию Vault с Kubernetes в *главе 7*. Советуем вам ее почитать, если вы хотите узнать больше о том, как это управление организовано в Kubernetes, и познакомиться с низкоуровневыми деталями интеграции Vault.

Наш опыт показывает, что при использовании внешних систем для конфигурации и конфиденциальных данных вопросы интеграции лучше делегировать платформе (в

как можно большей степени). Интеграцию с внешними системами, такими, как Vault, можно предлагать в качестве сервиса платформы, позволяющего предоставлять объекты `Secret` в виде томов или переменных окружения внутри Pod'ов. Сервис платформы инкапсулирует внешнюю систему, позволяя вашим приложениям использовать объекты `Secret`, не беспокоясь о том, как реализована эта интеграция. В целом применение таких сервисов делает приложения более простыми и способствует их стандартизации.

Реакция на события перепланирования

Kubernetes — это очень динамичная среда, в которой приложени перемещаются по разным причинам. Узлы кластера появляются и исчезают, они могут исчерпать свои ресурсы или даже выйти из строя. В ходе обслуживания кластера (например, при переходе на новую версию) администраторы платформы могут выгружать, блокировать или удалять узлы. Это лишь несколько примеров тех ситуаций, в которых ваши приложения могут быть принудительно остановлены и развернуты заново.

Какой бы ни была причина, динамичность Kubernetes может повлиять на доступность и работу вашего приложения. И хотя самым важным фактором здесь выступает его архитектура, это влияние можно минимизировать с помощью некоторых механизмов Kubernetes, которые будут рассмотрены в данном разделе. Для начала мы обсудим хуки жизненного цикла контейнера, которые срабатывают перед его остановкой. Затем речь пойдет о том, как контейнеры можно безопасно останавливать, что потребует обработки сигналов внутри приложения в ответ на события остановки. В завершение мы проведем обзор правил антиподобия Pod'ов — механизма, с помощью которого приложения можно распределять по зонам отказа. Как уже упоминалось ранее, подобные механизмы могут помочь *минимизировать* факторы нестабильности, но не устраняют саму возможность сбоев. Помните об этом при чтении данного раздела.

Хуки, срабатывающие перед остановкой контейнеров

Kubernetes может прерывать приложения по целому ряду причин. Если вам нужно выполнить действие перед тем, как будет остановлена работа контейнера, вы можете воспользоваться хуком его жизненного цикла. Kubernetes поддерживает два вида хуков. Хук жизненного цикла `exec` выполняет команду внутри контейнера, а хук `HTTP` шлет HTTP-запрос заданной конечной точке (роль которой обычно играет сам контейнер). Выбор хука зависит от ваших целей и от конкретных требований.

Контроллер Ingress Contour (**https://projectcontour.io**) — отличная иллюстрация возможностей, которые дают хуки жизненного цикла. Чтобы не отбрасывать уже отправленные клиентские запросы, у Contour есть хук, который просит Kubernetes выполнить команду перед остановкой контейнера. В следующем фрагменте кода, взятом из YAML-файла Развертывания Contour (листинг 14.5), показана конфигурация этого хука.

Листинг 14.5

```
# <... опущено ...>
    spec:
      containers:
      - command:
        - /bin/contour
        args:
          - envoy
          - shutdown-manager
        image: docker.io/projectcontour/contour:main
        lifecycle:
          preStop:
            exec:
              command:
                - /bin/contour
                - envoy
                - shutdown
# <... опущено ...>
```

Хуки жизненного цикла позволяют выполнить действия перед тем, как Kubernetes остановит ваш контейнер. Вы можете запускать команды или скрипты, размещенные внутри контейнера, но не являющиеся частью активного процесса. Один ключевой аспект, который необходимо учитывать, состоит в том, что эти хуки срабатывают только в ответ на штатные события жизненного цикла или перепланирования. К примеру, если узел выйдет из строя, они не будут выполнены. Более того, любое действие, выполненное в рамках такого хука, ограничено периодом безопасного завершения работы, который мы обсудим далее.

Безопасное завершение работы контейнеров

После выполнения хуков (если таковые имеются) Kubernetes инициирует процесс остановки контейнера, отправляя приложению сигнал `SIGTERM`. Таким образом контейнер узнает, что его скоро остановят. С этого момента начинается период завершения работы, который по умолчанию длится 30 секунд. Длительность периода можно изменить с помощью поля `terminationGracePeriodSeconds` в спецификации Pod'а.

Во время периода безопасного завершения работы приложение получает возможность выполнить любые необходимые действия. Это может быть сохранение данных, закрытие сетевых соединений, сброс файлов на диск и т. д. Закончив с этим, приложение должно прекратить работу, вернув код успешного завершения. Этот процесс проиллюстрирован на рис. 14.1. Как видите, kubelet шлет сигнал `SIGTERM` и ждет, пока контейнер не завершит свою работу в пределах выделенного ему времени.

Если приложение успевает прекратить выполнение за отведенный срок, процесс завершения работы заканчивается. В противном случае Kubernetes принудительно

останавливает приложение, отправляя ему сигнал SIGKILL. Эта процедура принудительного завершения показана в правой нижней части рис. 14.1.

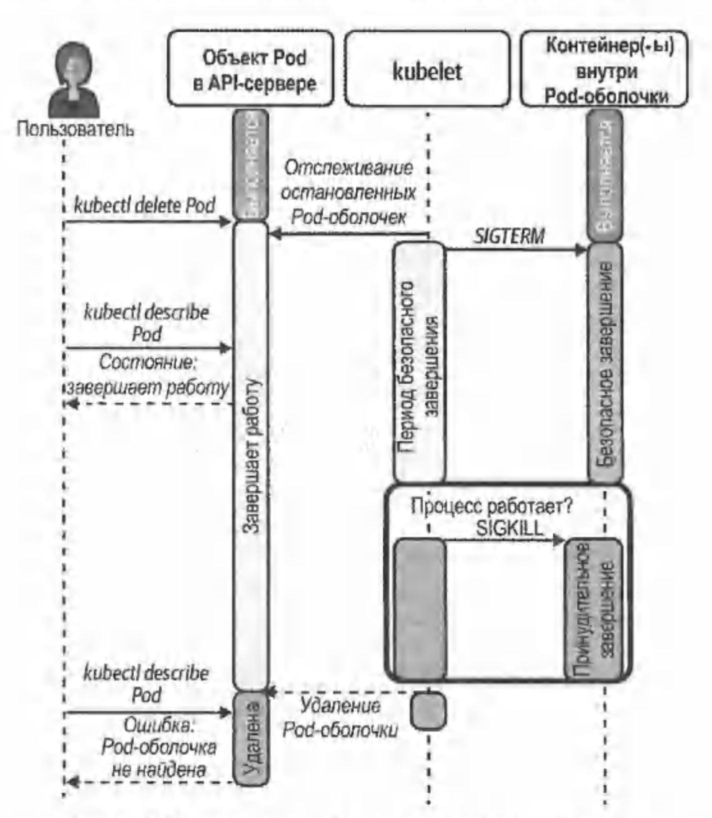


Рис. 14.1. Завершение работы приложения в Kubernetes. Сначала kubelet шлет приложению сигнал SIGTERM и ждет на протяжении периода безопасного завершения. Если по истечении этого периода процесс все еще работает, kubelet шлет сигнал SIGKILL, чтобы остановить его принудительно

Чтобы ваше приложение могло безопасно завершить свою работу, вы должны обработать сигнал SIGTERM. Каждый язык программирования или платформа разработки имеет свои механизмы для настройки обработчиков сигналов. Некоторые платформы могут даже сделать это за вас. В листинге 14.6 показано приложение на языке Go с обработчиком сигнала SIGTERM. Когда поступает сигнал, HTTP-сервер приложения останавливается.

Листинг 14.6

```
func main() {
        // Код инициализации приложения...
        httpServer := app.NewHTTPServer()

        // Создаем канал для отслеживания системных сигналов прерывания или
завершения
```

```
	// Используем буферизированный канал, так как этого требует пакет Signal
	shutdown := make(chan os.Signal, 1)
	signal.Notify(shutdown, os.Interrupt, syscall.SIGTERM)

	// Запускаем приложение и ждем возникновения ошибки
	errors := make(chan error, 1)
	go httpServer.ListenAndServe(errors)

	// Блокируем главный поток выполнения и ждем завершения работы
	select {
	case err := <-errors:
		log.Fatalf("http server error: %v", err)

	case <-shutdown:
	log.Printf("shutting down http server")
		httpServer.Shutdown()
	}
}
```

Если ваши приложения выполняются в Kubernetes, мы советуем настроить обработчики сигнала `SIGTERM`. Даже если вам не нужно предпринимать никаких действий для завершения работы, обработка этого сигнала улучшает поведение вашего приложения, ускоряя его остановку и освобождая тем самым ресурсы для других приложений.

Удовлетворение требований доступности

Хуки, срабатывающие перед остановкой контейнера, и безопасное завершение работы относятся к отдельному экземпляру (или реплике) вашего приложения. Если приложение горизонтально масштабируется, у вас, скорее всего, запущено несколько реплик, чтобы удовлетворить требования доступности. Наличие более одного экземпляра приложения может обеспечить повышенную отказоустойчивость. Например, если один из узлов кластера, на котором размещен экземпляр приложения, выйдет из строя, остальные реплики смогут его подменить. Тем не менее, существование нескольких реплик не поможет, если все они находятся в одной и той же зоне отказа.

Чтобы обеспечить распределение Pod'ов по разным зонам отказа, можно воспользоваться правилами антиподобия (англ. anti-affinity). Они позволяют сообщить планировщику Kubernetes о том, что Pod'ы должны размещаться с учетом ограничений, указанных в их определениях. В частности, вы можете попросить не размещать свои Pod'ы на узлах, которые уже выполняют реплики вашего приложения. Возьмем в качестве примера веб-сервер с тремя репликами. Чтобы все эти реплики не попали в одну и ту же зону отказа, вы можете воспользоваться правилом антиподобия Pod'ов, представленным в листинге 14.7. Данное правило просит планировщик по возможности распределять Pod'ы по разным зонам в соответствии с меткой `zones` узлов кластера.

Листинг 14.7

```
# ... <опущено> ...
      affinity:
        PodAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: "app"
                    operator: In
                    values:
                    - my-web-server
              topologyKey: "zone"
# ... <опущено> ...
```

Помимо правил антиподобия, Kubernetes предоставляет ограничения PodTopologySpread, которые позволяют еще лучше распределять Pod'ы по разным зонам отказа. Проблема антиподобия в том, что с его помощью невозможно гарантировать *равномерное* распределение Pod'ов по зонам. Вы можете либо "предпочесть", чтобы они развертывались с учетом ключа топологии, либо гарантировать, что в каждую зону отказа попадет только одна реплика.

Ограничения PodTopologySpread дают вам возможность попросить планировщик распределить ваши приложения. Аналогично правилам антиподобия, они применяются только к новым Pod'ам, которые нужно развернуть, и не имеют обратной силы. В листинге 14.8 приведен пример ограничения PodTopologySpread, которое приводит к распределению Pod'ов по разным зонам (в соответствии с меткой `zone`, которую содержат узлы). Если ограничение не удается соблюсти, Pod не развертывается.

Листинг 14.8

```
# ... <опущено> ...
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
# ... <опущено> ...
```

Если у вашего приложения есть несколько экземпляров, вы должны использовать эти механизмы размещения Pod'ов, чтобы повысить их устойчивость к инфраструктурным неполадкам. В противном случае Kubernetes может развернуть ваше приложение так, что вы не получите желаемого уровня отказоустойчивости.

Проверки состояния

Kubernetes использует множество сигналов для определения состояния и работоспособности приложений, размещенных на платформе. Что касается работоспособности, то Kubernetes известно о приложении только то, выполняется оно или нет. Это полезная информация, но ее, как правило, недостаточно для эффективного выполнения и администрирования приложений. В связи с этим предусмотрен механизм проверок (англ. probes). Проверки предоставляют Kubernetes дополнительные сведения о состоянии приложения.

Kubernetes предлагает три вида проверок, которые оценивают работоспособность, готовность и состояние запуска. Прежде чем подробно их обсуждать, давайте рассмотрим разные механизмы выполнения проверок, доступные для каждой из этих категорий:

- *Exec* — kubelet выполняет команду внутри контейнера. Проверка считается успешно пройденной, если команда возвращает нулевой код завершения. В противном случае kubelet считает контейнер неработоспособным.
- *HTTP* — kubelet отправляет HTTP-запросы конечной точке объекта Pod. Если код HTTP-ответа больше или равен 200, но меньше 400, проверка считается успешно пройденной.
- *TCP* — kubelet устанавливает TCP-соединение с контейнером на заданном порту. Если соединение установлено успешно, контейнер считается работоспособным.

Помимо общих механизмов, все проверки имеют один и тот же набор параметров, которые позволяют адаптировать их к вашим приложениям. В число этих параметров входят положительные и отрицательные пороговые значения, время ожидания и др. Каждый параметр подробно описан в документации Kubernetes, поэтому мы не будем на них останавливаться.

Проверки работоспособности

Проверки работоспособности помогают Kubernetes понять, исправно ли функционируют Pod'ы в кластере. На уровне узла kubelet постоянно проверяет Pod'ы, у которых настроены эти проверки. Когда проверка превышает порог отказа, kubelet считает Pod неисправным и инициирует его перезапуск. На рис. 14.2 приведена блок-схема, иллюстрирующая проверку работоспособности.

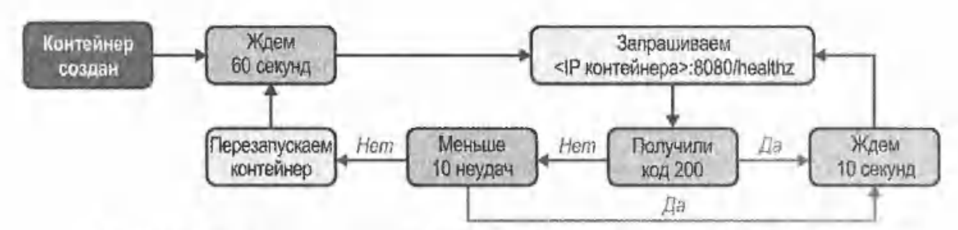


Рис. 14.2. Блок-схема проверки работоспособности на основе HTTP-запросов, проводимой раз в 10 секунд. Если проверка окажется неудачной 10 раз подряд, Pod будет считаться неисправным, и kubelet его перезапустит

Kubelet проверяет контейнер каждые 10 секунд, и, если последние 10 проверок завершились неудачей, контейнер перезапускается.



Учитывая, что неудачное прохождение проверок работоспособности приводит к перезапуску контейнера, мы обычно советуем реализовывать их так, чтобы они не проверяли внешние зависимости приложения. Делая свои проверки локальными, вы предотвращаете потенциальные каскадные сбои. Например, сервис, взаимодействующий с базой данных, не должен проверять ее "доступность" в рамках проверки работоспособности, так как перезапуск приложения почти наверняка не поможет решить проблему. Если приложению не удается соединиться с базой данных, оно должно перейти в режим для чтения или безопасным образом отключить функции, зависящие от БД. В качестве альтернативы приложение может провалить свою проверку готовности, о чем мы поговорим дальше.

Проверки готовности

Это, наверное, самый распространенный и важный вид проверок в Kubernetes, особенно для сервисов, обрабатывающих запросы. Kubernetes использует проверки готовности, чтобы понять, следует ли направлять трафик Сервиса Pod'ам. Следовательно, этот механизм позволяет приложению сообщить платформе о том, что оно готово к приему запросов.

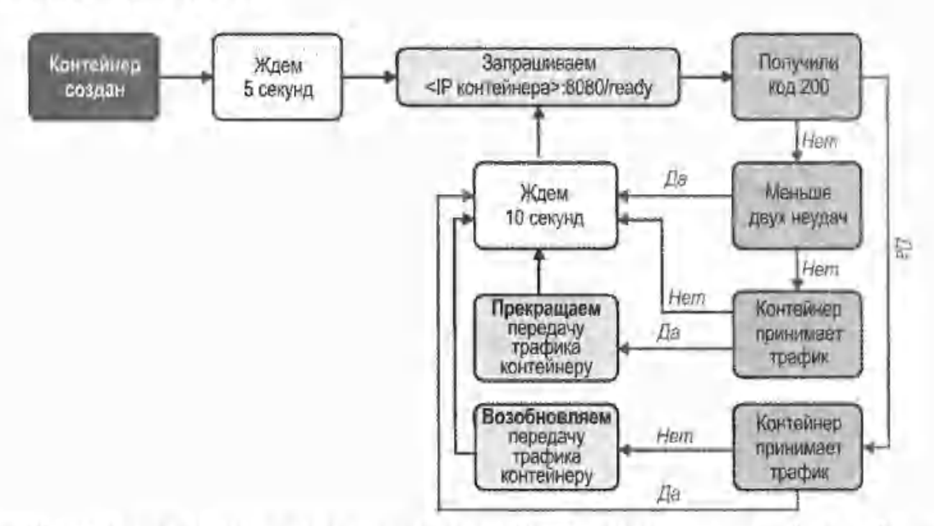


Рис. 14.3. Блок-схема проверки готовности на основе HTTP-запросов, проводимой раз в 10 секунд. Если проверка окажется неудачной два раза подряд, Pod будет объявлен неготовым и удален из списка готовых конечных точек

Как и в случае с проверками работоспособности, за проверку приложений и обновление состояния Pod'а в соответствии с результатами отвечает kubelet. Если Pod'у не удалось пройти проверку, платформа удаляет его из списка доступных конечных точек, фактически перенаправляя трафик к другим, готовым репликам. На рис. 14.3 показана блок-схема, которая объясняет принцип работы проверок готовности на основе HTTP-запросов. Длительность начальной задержки проверки составляет 5 секунд, период ожидания — 10 секунд. Запущенное приложение начинает при-

нимать трафик только после успешного прохождения проверки готовности. Если Pod "проваливает" эту проверку два раза подряд, платформа прекращает отправлять ей трафик.

При развертывании приложения, которое имеет вид сервиса, не забудьте настроить проверку готовности, чтобы избежать отправки запросов репликам, не готовым их обрабатывать. Эти проверки важны не только во время запуска, но и на протяжении всего времени жизни Pod'а. Они позволяют предотвратить маршрутизацию клиентского трафика к репликам, которые по каким-то причинам больше не готовы к его приему.

Проверки состояния запуска

Проверки работоспособности и безопасности доступны в Kubernetes, начиная с первой версии. Но с ростом популярности этой платформы сообщество выявило необходимость в реализации дополнительной проверки, относящейся к состоянию запуска. Эта проверка дает приложению дополнительное время на инициализацию. Подобно проверке работоспособности, в случае провала контейнер перезапускается. Однако после первого успешного выполнения проверка состояния запуска перестает проводиться, и дальше настает черед проверок работоспособности и готовности.

Если вам интересно, почему одной лишь проверки работоспособности недостаточно, давайте рассмотрим приложение, на инициализацию которого уходит в среднем 300 секунд. Действительно, вы могли бы задать проверку работоспособности, которая ждет 300 секунд и только затем останавливает контейнер. Она будет работать на этапе запуска. Но что будет происходить во время дальнейшей работы приложения? Если появятся неисправности, платформа перезапустит его только через 300 секунд! Значит это проблема, которую решает проверка состояния запуска. Она следит за приложением только во время его запуска. На рис. 14.4 показана блок-схема, иллюстрирующая принцип работы проверки состояния запуска, подобной той, которую мы только что обсудили. У нее есть порог отказа, 30 попыток, и период ожидания длительностью 10 секунд.



Рис. 14.4. Блок-схема проверки состояния запуска на основе HTTP-запросов, проводимой раз в 10 секунд. Если проверка возвращает успешный ответ, она отключается, и вместо нее начинают проводиться проверки работоспособности/готовности. Если она окажется неудачной 30 раз подряд, kubelet перезапустит Pod

Проверки состояния запуска могут иметь смысл для некоторых приложений, однако мы обычно советуем их избегать, если нет насущной необходимости. Мы считаем, что для большинства случаев вполне подходят проверки работоспособности и готовности.

Реализация проверок

Итак, мы рассмотрели разные виды проверок. Теперь давайте поговорим о том, как их следует использовать в приложениях. Нас в основном интересуют проверки работоспособности и готовности. Как мы знаем, если провалить первую, платформа перезапустит Pod, а провал второй приведет к тому, что Pod перестанет получать трафик. Учитывая разные исходы, мы считаем, что в большинстве случаев этим двум видам проверок, если они действуют совместно, необходимо предоставлять разные конечные точки или команды.

В идеале проверка работоспособности оказывается неудачной только при наличии проблемы, требующей перезапуска, такой, как взаимное блокирование или другая ситуация, которая мешает приложению выполнять свою работу. В приложениях, предоставляющих доступ к собственному HTTP-серверу, обычно реализована конечная точка, которая всегда возвращает код состояния 200. Пока этот HTTP-сервер исправно работает, приложение способно отвечать на запросы, поэтому его не нужно перезапускать.

Проверки готовности, в отличие от работоспособности, позволяют оценивать разные условия внутри приложения. Например, если во время запуска приложение "разогревает" свои внутренние кэши, то соответствующая конечная точка может возвращать false, пока кэш не будет готов. Еще одним примером является перегруженность, приложение может провалить проверку готовности, чтобы его меньше нагружали. Как можно себе представить, проверяемые условия зависят от конкретного приложения. Но в целом они имеют временный характер и в конечном счете исчезают.

Если подытожить, то мы обычно советуем применять проверки готовности в приложениях, обрабатывающих запросы, поскольку в другого рода приложениях, таких как контроллеры, плановые задания и т.д., они бессмысленны. Что касается проверок работоспособности, то их, по нашему мнению, следует рассматривать только в случаях, когда перезапуск приложения может помочь устранить проблему. И, наконец, мы предпочитаем избегать проверок состояния запуска, если в них нет крайней необходимости.

Запросы и лимиты на ресурсы Pod

Одна из главных обязанностей Kubernetes состоит в размещении приложений на узлах кластера. В процесс планирования среди прочего входит поиск подходящих узлов, у которых достаточно ресурсов для обслуживания приложения. Чтобы эффективно распределять приложения, планировщику Kubernetes необходимо знать, какие ресурсы им нужны. Обычно речь идет о центральном процессоре и памяти,

но иногда учитываются и другие типы ресурсов, такие как временное хранилище и даже пользовательские или расширенные объекты.

Когда платформа Kubernetes развертывает ваше приложение, ей требуется информация о ресурсах, которые нужны приложению, чтобы гарантировать их наличие на этапе выполнения. В конце концов, объем ресурсов платформы ограничен, и они разделяются между разными приложениями. Чтобы *использовать* эти ресурсы, приложение должно сообщить о своих требованиях.

В этом разделе мы обсудим запросы и лимиты на ресурсы, а также то, как они влияют на ваши приложения. Мы не станем углубляться в подробности о том, как платформа реализует эти запросы и лимиты, поскольку эта тема уже была рассмотрена в *главе 12*.

Запросы ресурсов

Это свойство позволяет указать минимальное количество ресурсов, необходимое для работы вашего приложения. Оно необходимо в большинстве приложений, развертываемых в Kubernetes. Так вы сможете гарантировать, что ваши приложения будут иметь доступ к запрошенным ресурсам на этапе выполнения. Если это свойство не указать, производительность вашего приложения может существенно снизиться, когда ресурсы узла начнут исчерпываться. Существует даже риск того, что ваше приложение будет принудительно остановлено, если узлу потребуется освободить память для других приложений. На рис. 14.5 показано принудительное завершение работы приложения в связи с тем, что другое приложение с запросами ресурсов начинает потреблять дополнительную память.



Рис. 14.5. Разделение памяти узла между Pod-оболочками Pod 1 и Pod 2. Изначально каждая Pod потребляет 200 МиБ из доступных 500 МиБ. Когда Pod 1 требуется дополнительная память, Pod 2 принудительно завершается, так как в ее спецификации нет соответствующих запросов. Pod 2 входит в аварийный цикл, так как ей не удается запуститься из-за нехватки памяти

Одна из основных трудностей при работе с запросами ресурсов состоит в подборе подходящего значения. Если вы развертываете существующее приложение, у вас уже могут быть данные, на основе которых можно определить его запросы ресурсов. Например, вам может быть известна фактическая ресурсоемкость приложения или конфигурация виртуальных машин, на которых оно размещается. В отсутствие

ретроспективных данных приходится делать обоснованные предположения и собирать информацию на протяжении какого-то времени. Вы также можете воспользоваться механизмом Vertical Pod Autoscaler (VPA), который может подсказать значения для запросов процессора и памяти или даже постепенно их регулировать. Подробней о VPA можно почитать в *главе 13*.

Лимиты на ресурсы

Это свойство позволяет указать максимальное количество ресурсов, которое может потребить ваше приложение. Вы можете спросить, зачем устанавливать искусственное ограничение? Чем больше ресурсов доступно, тем лучше, верно? Для некоторых приложений это действительно так, однако наличие неограниченного доступа к ресурсам может сделать производительность непредсказуемой, так как иногда в распоряжении Pod'а будут дополнительные ресурсы, а иногда — нет (если они нужны другим приложениям). Все еще серьезней, когда речь идет о памяти. Память — это нерегулируемый ресурс, и, чтобы его освободить, платформе придется принудительно останавливать Pod'ы, которые потребляют его максимально агрессивно.

При задании лимитов на ресурсы необходимо определиться с тем, следует ли их распространять на само приложение. Хороший пример — Java-приложения. Если у вас старая версия Java (JDK v8u131 и ниже), то лимит на память нужно устанавливать в том числе и в виртуальной машине Java (англ. Java Virtual Machine или JVM). В противном случае JVM не будет знать о лимите и попытается занять больше памяти, чем позволено. Вы можете сконфигурировать параметры памяти JVM с помощью переменной окружения `JAVA_OPTIONS`. Есть еще один способ (хоть и не всегда доступный): JVM можно обновить до более новой версии, которая уже умеет распознавать лимиты на память внутри контейнера. Если вы развертываете приложение с отдельной средой выполнения, подумайте о том, нужно ли ему вручную указывать лимиты на ресурсы, чтобы оно их распознало.

Лимиты также важны для тестирования производительности приложений. Как несложно себе представить, при каждом тестовом прогоне Pod'ы могут развертываться на разных узлах. Если к приложениям не применяются лимиты на ресурсы, результаты тестов могут сильно варьироваться, поскольку, при простое ресурсов узла приложение может превысить свои запросы ресурсов.

Обычно лимиты на ресурсы следует делать равными соответствующим запросам. Это гарантирует, что у приложения всегда будет один и тот же объем ресурсов, независимо от того, что происходит с соседними Pod'ами.

Журналирование приложений

Журнальные записи играют важную роль в диагностике и отладке приложений, как во время разработки, так и в условиях эксплуатации. Приложения, выполняемые в Kubernetes, должны как можно активней записывать информацию в потоки `STDOUT` и

STDERR. Это не только упрощает ваш код, но также является для платформы самым простым средством доставки журнальных записей в центральное хранилище. Данная тема была рассмотрена в *главе 9*, где мы обсудили разные стратегии, системы и инструменты для обработки журнальных записей. В этом разделе мы затронем некоторые вещи, которые необходимо учитывать при обдумывании механизма журналирования приложений. Вначале мы поговорим о том, что, собственно, нужно записывать в журнал. Затем сравним структурированные и неструктурированные журнальные записи. И в конце мы расскажем, как сделать журнальные записи более полезными за счет добавления в них контекстной информации.

Что записывать в журнал

Когда речь заходит о журналировании приложения, в числе первых необходимо решить вопрос о том, *что* включать в журнальные записи. Разработчики, как правило, имеют свое мнение на этот счет, но, как показывает наш опыт, они зачастую с этим переусердствуют. Если записывать в журнал слишком активно, вы можете получить огромный объем информации, в которой сложно найти что-то полезное. Если же записывать слишком редко, эффективная диагностика приложения окажется затруднительной. Как это часто бывает, необходимо найти баланс.

Сотрудничая с разработчиками приложений, мы выработали хороший практический подход, помогающий определить, нужно ли записывать в журнал те или иные данные. Просто спросите себя, может ли это сообщение повлечь за собой какие-либо действия? Если ответ положительный, сообщение следует записать. Если нет, сообщение, вероятно, окажется бесполезным.

Структурированные и неструктурированные журнальные записи

Журнальные записи приложения можно разделить на структурированные и неструктурированные. Как можно догадаться по названию, неструктурированная запись представляет собой текстовую строку без определенного формата. Это, пожалуй, самый распространенный вид журнальных данных, поскольку они не требуют никакого предварительного планирования со стороны разработчиков. У команды разработки могут быть общие руководящие принципы, но она имеет возможность записывать в журнал все, что угодно.

А вот у структурированных журнальных записей есть заранее определенные поля, которые необходимо заполнить при их сохранении. Они, как правило, имеют вид JSON-документа или пар "ключ – значение" (например, `time="2015-08-09T03:41:12-03:21" msg="hello World!" thread="13" batchId="5"`). Основным их преимуществом является то, что они хранятся в формате, доступном для автоматической обработки, что облегчает их получение и анализ. С другой стороны, структурированные журнальные записи обычно сложно читать, поэтому при реализации журналирования в своем приложении вы должны тщательно взвесить эти плюсы и минусы.

Контекстная информация в журнальных записях

Основное назначение журнальных записей — предоставлять сведения о том, что происходило внутри вашего приложения в тот или иной момент времени. Возможно, вы диагностируете проблему в активном приложении или проводите анализ первопричин какого-то происшествия. Для успешного выполнения таких задач журнальные записи, как правило, должны не только описывать произошедшее, но и выдавать какую-то контекстную информацию.

Возьмем в качестве примера платежное приложение. Когда в его конвейере обработки запросов обнаруживаются неполадки, попробуйте записать не только саму ошибку, но и окружающий ее контекст. Например, если ошибка произошла из-за того, что не удалось найти получателя платежа, добавьте в журнальную запись его ID или имя, ID пользователя, пытавшегося отправить платеж, сумму платежа и т.д. Эта контекстная информация облегчит диагностику проблем и поможет предотвратить их дальнейшем. Но избегайте попадания в журнал конфиденциальных данных. Нельзя допускать утечек пользовательских паролей или номеров банковских карт.

Предоставление метрик

Еще один важный источник сведений о поведении вашего приложения, помимо журнальных записей, — метрики. В случае их наличия вы можете сконфигурировать оповещения, чтобы знать, когда ваше приложение требует к себе внимания. Более того, со временем, накапливая метрики, вы можете обнаруживать тенденции, улучшения и ухудшения, вызванные выкатыванием новых версий вашего программного обеспечения. Этот раздел посвящен инструментированию приложений и некоторым из тех метрик, которые вы можете собирать, включая RED (Rate, Errors, Duration — частота, уровень ошибок, продолжительность), USE (Utilization, Saturation, Errors — использование, загруженность, ошибки) и метрики, характерные для отдельных приложений. Если вас интересуют компоненты платформы для мониторинга и дополнительная информация о метриках, обратитесь к *главе 9*.

Инструментирование приложений

В большинстве случаев платформа сама способна измерить и извлечь метрики о поведении вашего приложения. Такая информация, как использование процессора и памяти, IOPS диска (частота операций ввода/вывода) и другая предоставляется узлом, выполняющим приложения. Это полезные сведения, но вам стоит подумать об инструментировании своих приложений, чтобы самостоятельно предоставлять ключевые метрики.

Prometheus — одна из популярнейших систем мониторинга для платформ на основе Kubernetes, с которыми мы сталкиваемся в своей работе. Мы уже подробно рассмотрели ее и компоненты, из которых она состоит, в *главе 9*. В этом же разделе речь пойдет об инструментировании приложений с расчетом на Prometheus.

Prometheus извлекает метрики из вашего приложения, отправляя HTTP-запросы к конечной точке, указанной в конфигурации (обычно `/metrics`). Значит, ваше приложение должно эту конечную точку предоставить. Что еще более важно, ее ответ должен содержать метрики в формате Prometheus. В зависимости от вида программного обеспечения, которое вам нужно отслеживать, существуют два подхода к предоставлению метрик:

- *Встроенное инструментирование* — этот вариант подразумевает инструментирование с использованием клиентских библиотек Prometheus, чтобы предоставлением метрик занимался сам процесс приложения. Это отличное решение в случае, если исходный код приложения находится под вашим контролем.
- *Внешнее средство экспорта* — это дополнительный процесс, который выполняется рядом с вашим приложением, преобразует имеющиеся метрики и предоставляет их в формате, совместимом с Prometheus. Данный вариант лучше всего подходит для готового ПО, которое вы не можете инструментировать напрямую. Для его реализации обычно используется шаблон проектирования "sidecar контейнер". В качестве примеров можно привести NGINX Prometheus Exporter (**https://oreil.ly/g0ZCt**) и MySQL Server Exporter (**https://oreil.ly/SJOka**).

Библиотеки инструментирования Prometheus поддерживают четыре вида метрик: Counter (счетчики), Gauge (измерители), Histogram (гистограммы) и Summary (сводки). Счетчики — это метрики, значения которых могут только увеличиваться, а измерители могут изменяться в обоих направлениях. Гистограммы и сводки являются более сложными видами метрик. Гистограммы помещают собранные показатели в настраиваемые объектные контейнеры, которые можно использовать для вычисления квантилей (например, 95-й процентиль) на сервере Prometheus. Сводки похожи на гистограммы, но позволяют вычислять квантили на скользящем временном интервале на клиентской стороне. Эти виды метрик подробно описываются в документации Prometheus (**https://oreil.ly/epvwC**).

Инструментирование приложения с помощью библиотек Prometheus состоит из трех основных шагов. Давайте возьмем в качестве примера сервис на языке Go. Прежде всего, вам нужно запустить HTTP-сервер, чтобы система Prometheus могла собирать метрики. В библиотеке есть обработчик HTTP-запросов, который берет на себя преобразование метрик в формат Prometheus. Добавление этого обработчика выглядит примерно так:

```
func main() {
  // код приложения...
  http.Handle("/metrics",
    promhttp.HandlerFor(
      prometheus.DefaultGatherer,
      promhttp.HandlerOpts{},
  ))
  log.Fatal(http.ListenAndServe("localhost:8080", nil))
}
```

Затем вам нужно создать и зарегистрировать метрики. Например, если вы хотите сделать доступной метрику `Counter` под названием `items_handled_total`, воспользуйтесь кодом, приведенным в листинге 14.9.

Листинг 14.9

```
// создаем счетчик
var totalItemsHandled = prometheus.NewCounter(
  prometheus.CounterOpts{
    Name: "items_handled_total",
    Help: "Total number of queue items handled.",
  },
)

// регистрируем счетчик
prometheus.MustRegister(totalItemsHandled)
```

И, наконец, вам нужно обновлять свои метрики, чтобы они отражали то, что происходит в приложении. Продолжая пример с метрикой `Counter`, вы можете инкрементировать ее с помощью метода `Inc()`:

```
func handleItem(item Item) {
  // код для обработки элементов...

  // инкрементируем счетчик в ходе обработки элементов
  totalItemsHandled.Inc()
}
```

Библиотеки Prometheus делают процесс инструментирования приложения относительно простым. Более сложная задача — определение метрик, которые вам нужно предоставлять. В следующих разделах мы обсудим разные методы или философии, от которых можно отталкиваться при выборе метрик.

Метод USE

Метод USE, предложенный Бренданом Греггом (**http://www.brendangregg.com/usemethod.html**), предназначен для системных ресурсов. Он заключается в сборе таких показателей, как использование, загруженность и ошибки (Utilization, Saturation, Errors или USE) для каждого ресурса, который потребляет ваше приложение. К этим ресурсам обычно относятся процессор, память, диск и т. д. Они также могут включать ресурсы, существующие внутри приложения, такие как очереди, пулы потоков выполнения и т. д.

Метод RED

В отличие от USE, метод RED в первую очередь ориентирован на сами сервисы, а не на потребляемые ими ресурсы. Изначально предложенный Томом Уилки (**https://oreil.ly/sW3al**), он охватывает такие характеристики, как частота, уровень

ошибок и продолжительность (Rate, Errors и Durations) запросов, обрабатываемых сервисом. Метод RED может быть более подходящим для онлайн-сервисов, так как эти метрики предоставляют сведения о том, как вашу систему воспринимают конечные пользователи.

Четыре "золотых" сигнала

Еще одна философия, предложенная компанией Google в книге *Site Reliability Engineering* (**https://oreil.ly/iv1bJ**, O'Reilly), состоит в измерении четырех "золотых" сигналов для каждого сервиса: латентность (Latency), трафик (Traffic), ошибки (Errors) и насыщенность (Saturation). Вы, наверное, заметили, что эти сигналы похожи на метрики, собираемые в рамках метода RED, к которым еще добавлена насыщенность.

Метрики отдельных приложений

Методы USE и RED, равно как и четыре "золотых" сигнала охватывают метрики, применимые к большинству, если не ко всем приложениям. Но существует еще одна категория метрик, предоставляющая информацию о конкретном приложении. Например, сколько времени занимает добавление товара в корзину покупок? Или как долго устанавливается соединение между клиентом и агентом? Обычно эти метрики имеют отношение к ключевым показателям бизнес-эффективности (англ. Business Key Performance Indicators или KPI).

Какой бы метод вы ни выбрали, одним из важнейших факторов успеха вашего приложения является экспорт его метрик. Имея доступ к этим метрикам, вы можете создавать информационные панели для визуализации поведения своей системы, настраивать оповещения для уведомления дежурных о возникающих проблемах и проводить анализ тенденций с целью получения бизнес-аналитики, которая может поспособствовать развитию вашей организации.

Инструментирование сервисов для распределенной трассировки

Распределенная трассировка позволяет анализировать приложения, состоящие из множества сервисов. Таким образом можно исследовать путь обработки запроса по мере того, как он проходит через разные компоненты системы. Как уже отмечалось в *главе 9*, платформы на основе Kubernetes могут предоставлять распределенную трассировку в качестве одного из своих сервисов, используя такие системы, как Jaeger (**https://www.jaegertracing.io**) или Zipkin (**https://zipkin.io**). Но, аналогично мониторингу и метрикам, для применения этого подхода требуется инструментирование. В этом разделе мы рассмотрим процесс инструментирования сервисов с помощью Jaeger и OpenTracing (**https://opentracing.io**). Вначале мы покажем, как инициализировать трассировщик. Затем вы узнаете, как создавать спаны внутри сервиса. *Спан* — это именованная, ограниченная по времени операция, которая яв-

ляется составным элементом распределенного трейса. В конце речь пойдет о том, как передавать контекст трассировки от одного сервиса к другому. В своих примерах мы будем использовать язык Go и написанные на нем библиотеки, но подобные принципы применимы и к другим языкам программирования.

Инициализация трассировщика

Прежде чем создавать спаны внутри сервиса, мы должны инициализировать трассировщик. В рамках процесса инициализации мы должны установить параметры в соответствии с тем, в какой среде выполняется приложение. Трассировщику нужно знать имя сервиса, URL-адрес, по которому слать информацию о трассировке, и т.д. Для задания этих параметров мы рекомендуем использовать переменные окружения из клиентской библиотеки Jaeger. Например, имя сервиса можно указать с помощью переменной окружения `JAEGER_SERVICE_NAME`.

В ходе инициализации трассировщик можно не только сконфигурировать, но и интегрировать с библиотеками сбора метрик и журналирования. Первая служит для генерации метрик о состоянии трассировщика, например, о количестве собранных трейсов и спанов, а также о количестве успешно предоставленных спанов и пр. С помощью библиотек журналирования трассировщик генерирует журнальные записи при возникновении ошибок. Вы можете настроить его так, чтобы он записывал в журнал спаны, что довольно полезно на этапе разработки.

Чтобы инициализировать трассировщик Jaeger в сервисе на языке Go, нужно добавить код, подобный приведенному в листинге 14.10. В нашем примере мы выбрали Prometheus в качестве библиотеки сбора метрик и стандартную библиотеку журналирования из состава Go.

Листинг 14.10

```
package main

import (
        "log"

        jaeger "github.com/uber/jaeger-client-go"
        "github.com/uber/jaeger-client-go/config"
        "github.com/uber/jaeger-lib/metrics/prometheus"
)

func main() {
  // код инициализации приложения...

  metricsFactory := prometheus.New() ❶

  cfg := config.Configuration{} ❷
  tracer, closer, err := cfg.NewTracer( ❸
```

```
    config.Metrics(metricsFactory),
    config.Logger(jaeger.StdLogger),
  )
  if err != nil {
    log.Fatalf("error initializing tracer: %v", err)
  }

  defer closer.Close()

  // продолжаем main()
}
```

❶ Создание фабрики метрик Prometheus, с помощью которой Jaeger сможет генерировать метрики.

❷ Создание конфигурации по умолчанию для Jaeger без прописывания параметров прямо в коде (используются переменные окружения).

❸ Создание нового трассировщика на основе конфигурации, предоставление фабрики метрик и средства журналирования из стандартной библиотеки Go.

Инициализировав трассировщик, мы можем приступить к созданию спанов в нашем сервисе.

Создание спанов

Имея в своем распоряжении трассировщик, мы можем начать создавать спаны в нашем сервисе. Если предположить, что сервис находится где-то в середине цепочки обработки запросов, ему нужно десериализовать информацию о спане, поступающую от предыдущего сервиса, и создать дочерний спан. В нашем примере задействован HTTP-сервис, поэтому контекст спана передается в HTTP-заголовках. Код из листинга 14.11 извлекает контекст из заголовков и создает новый спан. Обратите внимание на то, что трассировщик, который мы инициализировали в предыдущем разделе, должен находиться в области видимости.

Листинг 14.11

```
package main

import (
        "github.com/opentracing/opentracing-go"
        "github.com/opentracing/opentracing-go/ext"
        "net/http"
)

func (s server) handleListPayments(w http.ResponseWriter, req *http.Request) {
        spanCtx, err := s.tracer.Extract( ❶
                opentracing.HTTPHeaders,
                opentracing.HTTPHeadersCarrier(req.Header),
```

```
        )
        if err != nil {
                // обработка ошибки
        }

        span := opentracing.StartSpan( ❷
                "listPayments",
                ext.RPCServerOption(spanCtx),
        )
        defer span.Finish()
}
```

❶ Извлечение контекстной информации из HTTP-заголовков.

❷ Создание нового спана на основе извлеченного контекста.

В ходе обработки запроса сервис может добавлять дочерние спаны в тот спан, который мы только что создали. Представьте, к примеру, что сервис вызывает функцию для выполнения SQL-запроса. Мы можем написать следующий код, чтобы создать для этой функции дочерний спан и назначить операции название `listPayments`:

```
func listPayments(ctx context.Context) ([]Payment, error) {
  span, ctx := opentracing.StartSpanFromContext(ctx, "listPayments")
  defer span.Finish()
  // выполняем SQL-запрос
}
```

Передача контекста

До сих пор мы создавали спаны внутри одного сервиса или процесса. Если в обработке запроса участвуют другие сервисы, нам нужно передавать контекст трейса по сети тому из них, который идет следующим. Как уже обсуждалось в предыдущем разделе, для этого можно использовать HTTP-заголовки.

Библиотеки OpenTracing предоставляют вспомогательные функции, позволяющие внедрять контекст в HTTP-заголовки. В листинге 14.12 приведен пример создания и отправки запроса с помощью HTTP-клиента из стандартной библиотеки Go.

Листинг 14.12

```
import (
    "github.com/opentracing/opentracing-go"
        "github.com/opentracing/opentracing-go/ext"

          "net/http"
)

// создаем HTTP-запрос
req, err := http.NewRequest("GET", serviceURL, nil)
```

```
if err != nil {
  // обработка ошибок
}

// внедряем контекст в HTTP-заголовки запроса
ext.SpanKindRPCClient.Set(span) ❶
ext.HTTPUrl.Set(span, url)
ext.HTTPMethod.Set(span, "GET")
span.Tracer().Inject( ❷
  span.Context(),
  opentracing.HTTPHeaders,
  opentracing.HTTPHeadersCarrier(req.Header),
)

// отправляем запрос
resp, err := http.DefaultClient.Do(req)
```

❶ Добавляет тег, чтобы назначить спан клиентской стороной обращения к сервису.

❷ Внедряет контекст спана в HTTP-заголовки запроса.

Как уже отмечалось ранее, инструментирование приложения для трассировки подразумевает инициализацию трассировщика, создание спанов внутри сервиса и передачу контекста другим сервисам. Существуют дополнительные функции, которые стоит исследовать, включая назначение тегов и журналирование. Если разработчики платформы предлагают трассировку в качестве одного из ее сервисов, вы теперь знаете, как ее реализовать.

Резюме

Работу приложений в Kubernetes можно оптимизировать разными способами. Для реализации большинства из них требуются время и усилия, но опыт показывает, что они являются ключом к выводу приложений на уровень эксплуатации. Добавляя приложения на свою платформу, не забывайте о рекомендациях, перечисленных в этой главе, включая внедрение конфигурации и конфиденциальных данных на этапе выполнения, задание запросов и лимитов на ресурсы, предоставление сведений о работоспособности приложения с помощью проверок и его инструментирование с использованием журнальных записей, метрик и трейсов.

ГЛАВА 15

Логистика доставки программного обеспечения

Цель вашей команды или компании никогда не должна сводиться исключительно к реализации платформы на основе Kubernetes (если только вы не поставщик или консультант!) Это может прозвучать странно для книги, всецело посвященной Kubernetes, но давайте на минуту отвлечемся. Любая компания занимается предоставлением товаров или услуг в своей основной области специализации. Это может быть платформа электронной торговли, система мониторинга SaaS или веб-сайт для оформления страховки. Такие платформы, как Kubernetes (и почти любой другой инструментарий) существуют для того, чтобы обеспечить профильную деятельность организации; многие команды, проектирующие и реализующие ИТ-решения, забывают об этом факте.

Данная глава написана именно в этом ключе. Мы сосредоточимся на процессе, с помощью которого разработчики доставляют свой программный код в реальное окружение на основе Kubernetes. Чтобы лучше всего продемонстрировать каждый существенный этап этого процесса, мы будем следовать знакомой многим модели конвейера.

Для начала будут рассмотрены некоторые факторы, которые нужно учитывать при сборке образов контейнеров (развертываемых нами ресурсов) из исходного кода. Если вы уже пользуетесь Kubernetes или другой контейнерной платформой, вам, наверное, уже знакомы некоторые из концепций, представленных в этом разделе, но мы постараемся обсудить здесь вопросы, о которых вы могли раньше не задумываться. Если вы только знакомитесь с контейнерами, это будет радикальный переход от того, как вы в настоящее время собираете свое программное обеспечение (WAR-файлы, исполняемые файлы Go и т. д.), к образам контейнеров и нюансам, связанным с их сборкой и обслуживанием.

Собранные ресурсы нужно где-то хранить. Мы обсудим реестры контейнеров (такие, как DockerHub, Harbor, Quay) и функции, которые, по нашему мнению, нужно учитывать при их выборе. Многие характеристики реестров связаны с безопасностью, и мы продемонстрируем некоторые из них, включая сканирование, обновление и подписание образов.

Наконец, мы уделим некоторое время непрерывной доставке и тому, как эта методология и связанные с ней инструменты пересекаются с Kubernetes. Мы поговорим о таких новых понятиях таких, как GitOps (развертывание путем синхронизации состояния кластера с репозиториями git), и более традиционном императивном подходе, основанном на конвейерах.

Даже если вы еще не используете Kubernetes, вам, скорее всего, придется обдумывать и/или решать все те общие задачи, которые мы только что упомянули (сборка, хранение ресурсов, развертывание). Вполне разумно предположить, что вокруг уже имеют опыт работы с существующим инструментарием. Нам очень редко встречаются организации, которые желают заново переработать весь процесс доставки программного обеспечения. Одна из вещей, на которые мы попытаемся обратить ваше внимание в этой главе, состоит в том, что в конвейере есть этапы, которые можно легко реализовать по-своему, выбирая на каждом из них наиболее эффективные подходы. По аналогии со многими темами, описанными в этой книге, положительные изменения вполне можно (и рекомендуется) вносить постепенно, оставаясь при этом сосредоточенным на предоставлении полезных продуктов и услуг.

Создание образов контейнеров

Прежде чем появились контейнеры, мы упаковывали приложения для развертывания на сервере в виде исполняемых файлов, сжатых архивов или обычного исходного кода. Они выполнялись либо самостоятельно, либо внутри сервера приложений. Нам также нужно было следить за тем, чтобы среда выполнения, в которой мы развертывали свое ПО, содержала нужные зависимости и конфигурацию, необходимую для его успешной работы.

В контейнерном окружении развертываемым ресурсом выступает образ контейнера. Он содержит не только сам исполняемый файл приложения, но также среду выполнения и все сопутствующие зависимости. Этот образ представляет собой набор слоев файловой системы в сочетании с некоторыми метаданными, которые вместе соответствуют спецификации образов OCI (Open Container Initiative). Это общепринятый стандарт в облачно-ориентированной экосистеме, благодаря которому процесс создания образов может быть реализован множеством разных способов (некоторые из них мы рассмотрим в следующих разделах), давая при этом артефакт, который можно выполнять во всех существующих контейнерных окружениях (подробней об этом можно почитать в *главе 3*).

Обычно, чтобы получить образ контейнера, нужно создать файл Dockerfile с его описанием и выполнить этот файл с помощью Docker Engine. Существует целая экосистема инструментов (каждый со своими собственными подходами), которые вы можете использовать для создания образов контейнеров в разных условиях. Если воспользоваться терминологией одного из таких инструментов под названием BuildKit (созданного компанией Docker), эту процедуру можно разделить на две части, внешнюю (frontend) и внутреннюю (backend). Внешняя часть — это метод описания высокоуровневого процесса, который предназначен для сборки образа (например, Dockerfile или Buildpack; к последнему мы еще вернемся в той главе). Внутренняя часть — это сама система сборки, которая берет описание, сгенерированное внешней частью, и выполняет перечисленные в нем команды в контексте файловой системы, чтобы создать образ.

Внутренней частью зачастую выступает демон Docker, что не всегда является подходящим вариантом. Например, если мы хотим выполнять процесс сборки в Kubernetes, нам нужно либо запустить демон Docker внутри контейнера (Docker внутри Docker), либо пробросить Unix-сокет Docker из операционной системы хоста в контейнер для сборки. Оба метода имеют свои недостатки, а второй из них может спровоцировать проблемы с безопасностью. В ответ были созданы альтернативные решения, такие как Kaniko. Kaniko использует ту же внешнюю часть (Dockerfile), но прибегает к другим методикам при создании образа, что позволяет этому инструменту работать внутри Pod'a Kubernetes. Выбирая способ создания образов, вы должны ответить на такие вопросы:

- Можно ли запускать сборщик с привилегиями администратора?
- Подходит ли нам проброс сокета Docker?
- Заботит ли нас необходимость выполнения демона?
- Хотим ли мы выполнять сборку в контейнере?
- Хотим ли мы, чтобы она выполнялась вместе с рабочими заданиями Kubernetes?
- Насколько активно мы планируем использовать кэширование слоев?
- Как наш выбор инструментария скажется на распределенной сборке?
- Какие *внешние* компоненты или механизмы определения образов мы хотим использовать? Какие из них поддерживаются?

В этом разделе мы сначала пройдемся по некоторым рекомендуемым и нерекомендуемым приемам создания контейнерных образов (Cloud Native Buildpacks), которые нам встречались. Надеемся, это поможет сделать создаваемые вами образы более качественными. Затем мы разберем альтернативные подходы и покажем, как все эти методики можно интегрировать в конвейер.

На ранних этапах разработки в организациях зачастую встает вопрос о том, кто должен отвечать за создание образов. Когда проект Docker только начинал приобретать популярность, его в основном принимали за инструмент, ориентированный на разработчиков. Исходя из нашего опыта, в мелких организациях разработчики все еще занимаются написанием файлов Dockerfile и определением процесса сборки образов своих приложений. Но, когда дело доходит до масштабного внедрения контейнеров (и Kubernetes), делегирование этих обязанностей отдельным разработчикам или группам разработки становится непосильным бременем. Во-первых, это требует дополнительных усилий со стороны разработчиков, отвлекая их от основной работы. Во-вторых, образы для эксплуатации получаются слишком разнородными, с минимальной стандартизацией или вообще без таковой.

В итоге наблюдается тенденция к абстрагированию процесса создания образов и передача ответственности за реализацию методов и инструментов source-to-image (которые заключаются в получении на вход репозитория с кодом и превращении его в образ контейнера, готовый к продвижению по конвейеру) от разработчиков к командам эксплуатации и обслуживания платформы. Мы обсудим этот подход далее в разделе "Cloud Native Buildpacks". Между тем, мы также часто наблюдаем ситуацию, когда группы, ответственные за платформу, проводят семинары и/или

помогают разработчикам с созданием образов и файлов Dockerfile. Это может быть эффективным решением на начальных этапах развития организации, но не в долгосрочной перспективе, учитывая соотношение числа разработчиков и тех, кто эксплуатирует платформу.

Плохая практика "золотых" базовых образов

На практике нам встречались несколько плохих подходов, которые, как правило, не дают командам разработчиков адаптировать свой образ мышления к методикам, возникшим в мире контейнеров и облачно-ориентированных технологий. Наверное, самый распространенный из них — понятие предопределенных, "золотых" образов. Суть в том, что в окружениях, предшествовавших появлению контейнеров, для использования внутри организации утверждались определенные базовые образы (например, сконфигурированный образ CentOS), и на нем должны были основываться любые приложения, предназначенные для внедрения в эксплуатацию. Обычно это делалось в целях безопасности, так как инструменты и библиотеки в образе проходили проверку. Но при переходе на контейнеры получалось так, что команды были вынуждены изобретать велосипед, скачивая полезные сторонние образы и переориентируя на них свои приложения и конфигурацию.

Это влечет за собой несколько смежных проблем. Во-первых, создание внутренней, адаптированной версии исходного образа требует дополнительных усилий. Во-вторых, на команду, которая обслуживает внутреннюю платформу, ложится ответственность за хранение и сопровождение этих внутренних образов. Поскольку данная ситуация может выйти из-под контроля (учитывая, сколько образов применяется в типичном окружении), этот подход обычно вредит безопасности, так как из-за дополнительной работы обновления проводятся редко (или вообще никогда).

Мы обычно рекомендуем объединить усилия с отделом безопасности и определить, каким именно целям служат "золотые" образы. Обычно актуальными оказываются несколько из следующих вариантов:

- гарантия наличия определенных агентов/программ;
- гарантия отсутствия уязвимых библиотек;
- гарантия того, что пользовательские учетные записи имеют корректные права доступа.

Вместо этого мы можем разобраться в причинах, стоящих за данными ограничениями, и воплотить соответствующие требования в собственном инструментарии, который будет работать в рамках конвейера и отклонять несовместимые образы (или уведомлять об их наличии). Это позволит поддерживать желаемый уровень безопасности, а команды смогут пользоваться готовыми образами, разработанными сообществом (экономя усилия, ушедшие на их создание). В разделе "Реестры образов" данной главы будет подробно рассмотрен пример одного из таких рабочих процессов.

Одна из самых важных причин использования базового системного образа состоит в том, что в случае возникновения каких-либо проблем работники организации

должны уметь их диагностировать. Но, если копнуть немного глубже, это не так полезно, как может показаться. Потребность в диагностике конкретных проблем путем выполнения команд в активных контейнерах должна возникать крайне редко, но даже в таких случаях отличия между операционными системами, основанными на Linux, довольно тривиальные для принятия необходимых мер. К тому же, все больше и больше приложений упаковывается в ультралегковесные, пустые или бездистрибутивные образы для снижения накладных расходов внутри контейнера.

Попыток адаптировать все исходные/сторонние образы к своей базовой системе (или системам) следует избегать по причинам, описанным в этом разделе. Но мы вовсе не утверждаем, что поддержка внутреннего набора проверенных базовых образов нецелесообразна. Такие образы могут служить отличной основой для ваших собственных приложений, и некоторые факторы, которые стоит учитывать при их создании, будут рассмотрены в следующем разделе.

Выбор базового образа

Базовый образ контейнера определяет низкоуровневые слои, на которых будет основан образ приложения. Он играет ключевую роль, так как в его состав обычно входят библиотеки и инструменты операционной системы, которые будут частью итогового контейнерного образа. Если при выборе базового образа не проявить осмотрительность, он может стать источником лишних библиотек и инструментов, которые не только сделают ваш собственный образ раздутым, но также послужат потенциальными уязвимостями безопасности.

В зависимости от зрелости организации и предпринимаемых ею мер безопасности, у вас может не быть возможности выбирать свои базовые образы. Мы сотрудничали со многими организациями, в которых отдельные группы разработчиков отвечали за подготовку и сопровождение набора утвержденных базовых образов, которые были предназначены для разных отделов. Тем не менее, если у вас есть выбор, или вы один из тех, кто эти образы проверяет, руководствуйтесь следующими рекомендациями:

- Убедитесь в том, что образы опубликованы поставщиком с хорошей репутацией. Не стоит доверять случайному пользователю DockerHub. В конце концов, эти образы послужат основой для большинства или даже всех ваших приложений.
- Разберитесь в цикле обновлений и отдавайте предпочтение образам, которые обновляются непрерывно. Как уже упоминалось, базовый образ обычно содержит библиотеки и инструменты, к которым нужно применять исправления при обнаружении новых уязвимостей.
- Отдавайте предпочтение образам с открытой спецификацией или процессом сборки. Обычно речь идет о файле Dockerfile, который можно проанализировать и понять на основании этого, как собирается образ.
- Избегайте образов с ненужными вам инструментами или библиотеками. Старайтесь использовать минимальные, компактные образы, которые ваши разработчики могут взять за основу в случае необходимости.

Как показывает наш опыт, в большинстве случаев создавать свои собственные образы лучше на основе пустых (scratch) или бездистрибутивных образов, так как оба варианта воплощают в себе перечисленные ранее принципы. Пустой образ не содержит абсолютно ничего и может оказаться самым компактным из возможных вариантов, если поместить в него простой статический исполняемый файл. Но, если вам нужны сертификаты корневого ЦС или какие-то другие ресурсы, то у вас могут возникнуть проблемы. Их можно скопировать, но это требует дополнительного внимания. В большинстве случаев мы рекомендуем бездистрибутивные базовые образы, так как они уже содержат разумный набор заранее созданных учетных записей (nonroot, nobody и т. д.) и минимальное число необходимых библиотек, список которых зависит от того, какого рода базовый образ вы выбрали. Существует несколько версий бездистрибутивных образов, предназначенных для разных языков программирования.

В следующих нескольких разделах мы продолжим обсуждение рекомендуемых подходов, начиная с того, насколько важно указать подходящего пользователя, от имени которого будет выполняться ваше приложение.

Выбор пользователя для выполнения контейнера

Учитывая модель изоляции контейнеров (в основном тот факт, что контейнеры работают на одном и том же ядре Linux), выбор пользователя, от имени которого они запускаются, имеет важные последствия, о которых не подозревают некоторые разработчики. В большинстве случаев, если не указать пользователя для запуска контейнера, процесс выполняется от имени администратора (root). Это создает дополнительные направления атак на контейнер, что является проблемой. Например, если злоумышленнику удастся взломать приложение и выйти за пределы контейнера, он может получить администраторский доступ к хосту, на котором тот выполняется.

При создании собственного контейнерного образа очень важно правильно выбрать пользователя, от имени которого будет запускаться контейнер. Нужны ли приложению администраторские привилегии? Полагается ли приложение на содержимое файла /etc/passwd? Нужно ли добавить в образ контейнера пользователя nonroot? Отвечая на эти вопросы, не забудьте указать в конфигурации образа учетную запись, которая будет использоваться для его выполнения. Если ваш образ собирается с помощью Dockerfile, для этого можно воспользоваться директивой `USER`, как показано в следующем примере, в котором исполняемый файл my-app запускается от имени пользователя nonroot, входящего в одноименную группу (в бездистрибутивных образах эта конфигурация применяется по умолчанию):

```
FROM gcr.io/distroless/base
USER nonroot:nonroot
COPY ./my-app /my-app
CMD ["./my-app", "serve"]
```

И хотя учетную запись можно указать и в манифестах развертывания Kubernetes, ее определение в рамках спецификации контейнерного образа имеет свои преимущества, так как ваш образ оказывается самодокументированным. В результате появля-

ется гарантия, что все разработчики, запускающие контейнер в своих локальных окружениях, делают это от имени одного и того же пользователя из одной и той же группы.

Явное определение версий пакетов

Если ваше приложение использует внешние пакеты, вы, вероятно, устанавливается их с помощью диспетчера пакетов, такого как apt, yum или apk. В ходе создания образа контейнера необходимо явно определить или указать версии этих зависимостей. Например, в следующем примере показано приложение, зависящее от imagemagick. Инструкция `apk` в Dockerfile явно определяет версию `imagemagick`, совместимую с этим приложением:

```
FROM alpine:3.12
<...опущено...>
RUN ["apk", "add", "imagemagick=7.0.10.25-r0"]
<...опущено...>
```

Если версию не указать, существует риск получения других пакетов, которые могут нарушить работу вашего приложения. Поэтому всегда указывайте версии пакетов, которые устанавливаются в вашем контейнерном образе. Этим вы гарантируете возможность воспроизведения процесса его сборки и совместимость его пакетов.

Образы для сборки и выполнения приложений

С помощью контейнеров разработчики могут не только упаковывать свои приложения, но и собирать их. Контейнеры могут служить строго очерченным окружением, которое можно, к примеру, описать в Dockerfile. Это удобно, так как разработчикам не нужно устанавливать инструментарий для сборки в своей системе. Что еще более важно, контейнеры дают возможность организовать стандартизированную среду сборки для всей команды разработки и ее систем непрерывной интеграции (англ. Continuous Integration или CI).

Использование контейнеров для сборки приложений может быть полезным приемом, но необходимо различать сборочные и исполняемые образы контейнеров. Сборочный образ содержит все инструменты и библиотеки, необходимые для компиляции приложения, тогда как в исполняемом образе находится само приложение, которое мы развертываем. Например, если взять программное обеспечение на языке Java, сборочный образ может содержать JDK, Gradle/Maven и весь наш инструментарий для компиляции и тестирования. В то же время в исполняемый образ может входить только среда выполнения Java и наше приложение.

Учитывая, что на этапе выполнения приложениям, как правило, не нужны средства сборки, эти инструменты не должны попасть в исполняемый образ. В результате контейнер быстрее распространяется и менее подвержен атакам. Если ваши образы собираются с помощью системы Docker, вы можете задействовать ее функцию поэтапной сборки, чтобы отделить сборочный образ от исполняемого. В листинге 15.1 показано содержимое Dockerfile для приложения на Go. На этапе сборки использу-

ется образ `golang`, содержащий инструменты разработки на Go, а на этапе выполнения за основу берется пустой образ, в который входит лишь исполняемый файл приложения.

Листинг 15.1

```
# Этап сборки
FROM golang:1.12.7 as build ❶

WORKDIR /my-app

COPY go.mod . ❷
RUN go mod download
COPY main.go .
ENV CGO_ENABLED=0
RUN go build -o my-app

# Этап развертывания
FROM gcr.io/distroless/base ❸
USER nonroot:nonroot ❹
COPY --from=build --chown=nonroot:nonroot /my-app/my-app /my-app ❺
CMD ["/my-app"]
```

❶ Главный образ `golang` содержит все инструменты сборки для Go, которые не требуются на этапе выполнения.

❷ Вначале мы копируем файл go.mod и выполняем скачивание, чтобы закэшировать этот этап на случай, если код изменится, а зависимости останутся прежними.

❸ Мы можем использовать бездистрибутивный исполняемый образ (`distroless`), чтобы получить минимальную основу без дополнительных зависимостей.

❹ Мы хотим по возможности выполнять свои приложения от имени обычного пользователя (`nonroot`).

❺ Из этапа сборки на этап развертывания копируется только скомпилированный файл (`my-app`).



Контейнеры выполняют всего один процесс и обычно не содержат супервизор или систему инициализации. В связи с этим вам нужно самостоятельно позаботиться о корректной обработке сигналов и выгрузке или назначении "родителей" для покинутых процессов. Существует несколько минимальных скриптов инициализации, способных удовлетворить эти требования и выполнить начальную конфигурацию экземпляра вашего приложения.

Cloud Native Buildpacks

Еще один метод сборки контейнерных образов состоит в использовании инструментов, которые анализируют исходный код приложения и автоматически генерируют соответствующий образ. Подобно инструментам, ориентированным на от-

дельные приложения, этот подход существенно облегчает задачу разработчикам, так как им не нужно создавать и сопровождать файлы Dockerfile. Проект Cloud Native Buildpacks (облачно-ориентированные пакеты сборки) является реализацией этого подхода, и его общий принцип работы показан на рис. 15.1.

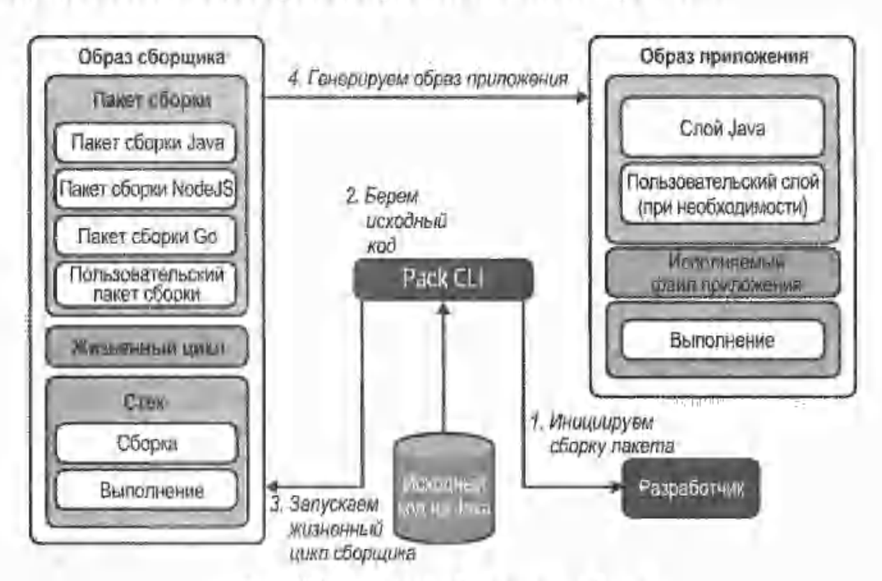


Рис. 15.1. Принцип работы пакета сборки

Cloud Native Buildpacks (CNB) — это реализация пакетов сборки (технологии, которую Heroku и Cloud Foundry использовали годами, чтобы упаковывать приложения для своих платформ), ориентированная на контейнеры. CNB упаковывает приложения в контейнерные образы OCI, готовые к выполнению в Kubernetes. Чтобы создать образ, CNB анализирует ваш исходный код и выбирает подходящий пакет сборки. Например, пакет Go выполняется, если в вашем исходном коде есть файлы, содержащие код на Go. Аналогично, пакет сборки Maven (Java) выполняется, если CNB находит файл pom.xml. Все это происходит "под капотом", и разработчики могут инициировать этот процесс с помощью инструмента командной строки под названием pack. Замечательная черта этого подхода — пакеты сборки имеют узкий охват, что позволяет создавать высококачественные образы, соответствующие рекомендациям.

Помимо облегчения процесса разработки и внедрения платформы, пользовательские пакеты сборки позволяют обеспечить соблюдение политик и нормативно-правовых требований, а также стандартизировать контейнерные образы, выполняемые на вашей платформе.

В целом, предоставление механизма для сборки контейнерных образов из исходного кода может быть стоящей затеей. Более того, наш опыт показывает, что польза от такого решения растет по мере увеличения размера организации. В конце концов, разработчики хотят сосредоточиться на написании полезных приложений, а не на их контейнеризации.

Реестры образов

Если вы уже применяете контейнеры, у вас, скорее всего, есть предпочтения относительно реестра. Это одно из основных условий использования Docker и Kubernetes, поскольку нам нужно где-то хранить образы, которые собираются на одном компьютере и затем развертываются на многих других (либо отдельных, либо входящих в кластер). OCI предлагает стандартную спецификацию не только для образов, но и для операций с реестрами (для обеспечения взаимодействия). У нее есть множество коммерческих и открытых реализаций, большинство из которых обладают общим набором возможностей. Чаще всего реестры образов состоят из трех компонентов: сервера (для логики взаимодействия с пользователем и API-интерфейса), хранилища BLOB-объектов (для самих образов) и базы данных (для метаданных о пользователях и образах). Хранилище, как правило, можно указать в конфигурации, и это может повлиять на проектирование архитектуры своего реестра. Чуть позже мы к этому еще вернемся.

В этом разделе мы рассмотрим некоторые из важнейших возможностей, которые предлагают реестры, и способы их внедрения в ваш конвейер. Мы не станем глубоко анализировать конкретные реализации реестров, так как в целом они имеют аналогичные функции, однако существуют ситуации, в которых можно склоняться к тому или иному решению в зависимости от имеющейся конфигурации или требований.

Если вы уже используете хранилище артефактов, такое как Artifactory или Nexus, вам, возможно, имеет смысл остановить свой выбор на их услугах по размещению образов, так как это упростит администрирование. Аналогично, если ваши окружения строго ориентированы на облачные платформы, вам может быть выгодно, с финансовой точки зрения, использовать реестр своего облачного провайдера такой, как AWS Elastic Container Registry (ECR), Google Container Registry (GCR) или Azure Container Registry (ACR).

Топология, архитектура и зоны отказа ваших окружений и кластеров — это тоже ключевые факторы, которые нужно учитывать при выборе реестра. Вы можете разместить по одному реестру в каждой зоне отказа, чтобы обеспечить высокую доступность. При этом вам необходимо решить, какое хранилище BLOB-объектов вам нужно, централизованное или распределенное по всем регионам с репликацией между экземплярами. Возможность репликации поддерживается большинством реестров и состоит в том, что образ, загруженный вами в один из группы реестров, автоматически копируется во все экземпляры этой группы. Даже если эта функция не поддерживается выбранным вами реестром напрямую, ее довольно легко реализовать с помощью системы конвейеров (такой как Jenkins) и веб-хуков, которые срабатывают при каждой загрузке образа.

Выбор между одним реестром и множеством реестров зависит от того, какую пропускную способность вам нужно поддерживать. В организациях с тысячами разработчиков, инициирующих сборку исходного кода и образов при каждой фиксации, количество параллельно выполняемых операций (скачиваний и загрузок) может

быть существенным. Поэтому важно понимать, что реестр образов, несмотря на свою ограниченную роль в конвейере, является ключевым компонентом не только для развертывания, но и для процесса разработки. Чтобы достичь высокого уровня доступности, он нуждается в мониторинге и сопровождении, как и любой другой важный механизм.

Многие реестры рассчитаны на то, чтобы их можно было легко использовать внутри кластера или контейнерного окружения. Этот подход (к которому мы еще вернемся далее в разделе "Непрерывная доставка") обладает множеством преимуществ. Прежде всего, мы имеем возможность пользоваться всеми этими компонентами и стандартами внутри Kubernetes, чтобы обеспечивать работу сервисов, делая их доступными для обнаружения и легко настраиваемыми. Очевидный недостаток здесь в том, что мы теперь зависим от сервиса внутри кластера, который предоставляет образы для запуска новых сервисов внутри того же кластера. Реестры чаще востребованы в разделяемых кластерах и имеют систему обеспечения отказоустойчивости, которая выполняет резервное копирование и гарантирует, что как минимум некоторые из экземпляров реестра всегда будут готовы к обработке запросов.

Нам также часто встречаются реестры, которые размещаются за пределами Kubernetes и считаются, скорее, отдельным компонентом для начальной конфигурации, необходимым для всех кластеров. Так обычно происходит, когда организация уже использует экземпляр Artifactory или другого реестра, и адаптирует его к размещению образов. В этом контексте также часто применяются облачные реестры, хотя вам нужно учитывать их гарантии доступности (так как здесь актуальны те же вопросы касательно топологии) и потенциальное увеличение задержек.

В следующих подразделах мы рассмотрим некоторые из наиболее распространенных вопросов, возникающих при выборе и эксплуатации реестра. Все они имеют отношение к безопасности, так как в основе защиты нашей цепочки доставки программного обеспечения лежат развертываемые нами артефакты (образы). Вначале мы обсудим сканирование уязвимостей и то, как гарантировать отсутствие в наших образах известных проблем безопасности. Затем будет описана широко используемая процедура карантина, которая позволяет выполнять эффективную доставку внешних/сторонних образов в наши окружения. И в конце мы поговорим о том, как сделать образ доверенным и как его подписать. В этом заинтересованы многие организации, но реализация в имеющихся инструментах и методах все еще не достигла зрелого состояния.

Сканирование уязвимостей

Сканирование образов на предмет известных уязвимостей — ключевая функция большинства реестров. Обычно этот процесс вместе с базой данных распространенных уязвимостей и рисков (англ. Common Vulnerabilities and Exposures или CVE) делегируют стороннему компоненту. Одним из таких компонентов является Clair — популярное решение с открытым исходным кодом, которое в большинстве случаев поддается расширению, чтобы удовлетворить специфические требования.

У любой организации есть свое представление о том, что такое приемлемый риск в контексте рейтингов CVE. Реестры обычно предоставляют механизмы, позволяющие запретить скачивание образов, рейтинг CVE которых превышает определенный порог. Кроме того, возможность добавления CVE в "белый" список может пригодиться на случай, если проблема существует, но не относится к вашему окружению, или для тех CVE, которые считаются приемлемым риском и/или не имеют доступных исправлений.

Это статическое сканирование на начальном этапе загрузки может послужить хорошей отправной точкой, но что, если уязвимости со временем обнаружатся в образах, которые уже присутствуют в нашем окружении. Для обнаружения таких изменений можно запланировать регулярное сканирование, но тогда нам нужно выработать план обновления и замены образов. У вас может возникнуть соблазн автоматизировать исправление и загрузку обновленных образов, и для этого предусмотрены решения, которые пытаются всегда поддерживать образы в актуальном состоянии. Но это может привести к проблемам, так как изменения, возникающие в результате обновления образов, могут оказаться несовместимыми и/или нарушить работу приложения. К тому же, такие автоматические системы могут работать вне вашего внутреннего процесса развертывания изменений, что сделает их аудит внутри окружения проблематичным. Даже блокирование скачивания образов (описанное ранее) может создать проблемы. Представьте, что в образе важного приложения обнаружилась новая уязвимость, а скачивание почему-то запрещено. В результате не удастся скачать образ приложения, и, если его экземпляры планируются для развертывания на новых узлах, это может плохо сказаться на его доступности. Как уже неоднократно отмечалось в этой книге, при реализации любого решения обязательно необходимо искать нужный вам баланс (в данном случае между безопасностью и доступностью) и принимать обоснованные и хорошо задокументированные решения.

Вместо описанного автоматического применения исправлений сканирование уязвимостей чаще сочетают с системами оповещения и/или мониторинга, которые передают информацию эксплуатационным командам и отделам безопасности. Реализация таких оповещений может варьироваться в зависимости от возможностей выбранного вами реестра. Некоторые реестры можно сконфигурировать так, чтобы по завершении сканирования они вызывали веб-хуки, передавая им параметры со сведениями об уязвимых образах и обнаруженных CVE. Другие могут предоставлять ту же информацию в виде набора метрик, на основе которых можно генерировать оповещения с использованием стандартных инструментов (подробней о метриках и средствах оповещения можно почитать в *главе 9*). И хотя этот метод требует активного ручного вмешательства, он позволяет получить более четкое представление о состоянии образов в вашем окружении и лучше контролировать то, как и когда происходит применение исправлений.

Имея информацию об уязвимостях в нашем образе, мы можем принять решение о том, исправлять его или нет (и, если да, то когда), принимая во внимание потенциальные последствия. Если нам нужно применить к образу исправления и обновления, мы можем инициировать процессы обновления, тестирования и развертывания

посредством наших обычных конвейеров. Это позволяет добиться полной прозрачности и аудитоспособности, а также гарантировать то, что все внесенные изменения пройдут наши стандартные процедуры. CI/CD и модели развертывания будут подробно рассмотрены далее в этой главе.

Статическое сканирование уязвимостей, обсуждаемое в этом подразделе, является частью цепочки доставки программного обеспечения во многих организациях, но это лишь один слой стратегии углубленной защиты, обеспечивающей безопасность контейнеров. Образы могут скачивать вредоносное содержимое уже после развертывания, а контейнерные приложения могут быть скомпрометированы или взломаны во время работы. В связи с этим обязательно нужно реализовать какого-то рода сканирование на этапе выполнения. В упрощенном виде это может выглядеть как периодическое сканирование файловой системы активного контейнера, позволяющее убедиться в том, что в нем после развертывания не появилось никаких уязвимых исполняемых файлов и/или библиотек. Но, если вам нужна защита понадежней, вы должны ограничить поведение контейнера и действия, которые он способен выполнять. Это позволит избежать игры в кошки-мышки, которая неизбежно возникает, если пытаться искать и исправлять CVE, вместо того чтобы сосредоточиться на возможностях, которыми должно обладать контейнерное приложение. Сканирование на этапе выполнения — это обширная тема, которую мы не можем рассмотреть здесь во всех деталях. Если вам хочется узнать больше, ознакомьтесь с такими инструментами, как Falco (**https://falco.org**) и пакет Aqua Security (**https://github.com/aquasecurity**).

Процедура карантина

Как уже упоминалось, большинство реестров дают возможность сканировать образы на предмет известных уязвимостей и блокировать их скачивание. Но, прежде чем разрешить использование образа, к нему можно предъявить дополнительные требования. Нам также встречались ситуации, когда разработчики не могут скачивать образы из Интернета и вынуждены пользоваться внутренним реестром. Оба подхода можно реализовать с помощью конфигурации с несколькими параллельными реестрами и процедурой карантина, описанной далее.

Для начала разработчикам можно предоставить портал самообслуживания, где они могли бы запрашивать образы. Для этого хорошо подойдет нечто вроде ServiceNow или задания Jenkins, и нам неоднократно встречались такие решения. Все большей популярностью пользуются чат-боты способные обеспечить более удобную интеграцию. Как только разработчики запрашивают образ, тот автоматически скачивается в карантинный реестр, где его можно проверить, и где конвейер в специальных окружениях может подтвердить, что образ соответствует определенным критериям.

После прохождения проверок образ может быть подписан (это необязательно, см. далее раздел "Подписание образов") и загружен в одобренный реестр. Система также может оповестить разработчика (с помощью чат-бота или за счет обновления заявки/задания) о том, что образ был принят (или отклонен, с пояснением причин). Весь процесс показан на рис. 15.2.



Рис. 15.2. Процедура карантина

Этот процесс можно использовать в сочетании с контроллерами допуска, чтобы в кластере могли выполняться только образы, которые были подписаны или взяты из определенного реестра.

Подписание образов

Вопрос безопасности цепочки доставки становится более актуальным по мере того, как приложения обрастают все большим числом внешних зависимостей, будь то библиотеки с кодом или образы контейнеров.

Одна из мер безопасности, часто упоминаемая при обсуждении образов, — понятие цифровой подписи. Говоря простым языком, тот, кто публикует образ, может его подписать с помощью криптографических средств, сгенерировав из него хеш и привязав этот хеш к нему перед загрузкой в реестр. Благодаря этому пользователи образа смогут подтвердить его подлинность, сравнив подписанный хеш с открытым ключом издателя.

Такая процедура привлекательна тем, что позволяет создать образ в начале нашей цепочки доставки ПО и подписывать его после каждого этапа конвейера. Например, мы можем его подписать после проведения тестирования и затем снова, после того как команда, отвечающая за подготовку новых выпусков, утвердит его для развертывания. Затем мы можем принять этот образ в эксплуатацию или отклонить его в зависимости от того, был ли он подписан различными участниками процесса, которых мы укажем. Это не только гарантирует, что образ получил необходимое одобрение, но и подтверждает, что в эксплуатацию попадет именно он, а не какой-то другой. Общий принцип работы показан на рис. 15.3.

Ведущий проект в этой области — Notary, который изначально был разработан компанией Docker на основе системы The Update Framework (TUF), предназначенной для безопасной доставки обновлений программного обеспечения.

Несмотря на преимущества процедуры подписания образов, мы не наблюдаем ее активного внедрения в реальных проектах, и тому есть несколько причин. Во-первых, Notary состоит из нескольких компонентов, включая сервер и набор баз данных. Их нужно дополнительно устанавливать, настраивать и поддерживать. К тому же, система Notary должна быть сконфигурирована с расчетом на высокую доступность и устойчивость, так как она является частью важного процесса развертывания программного обеспечения.

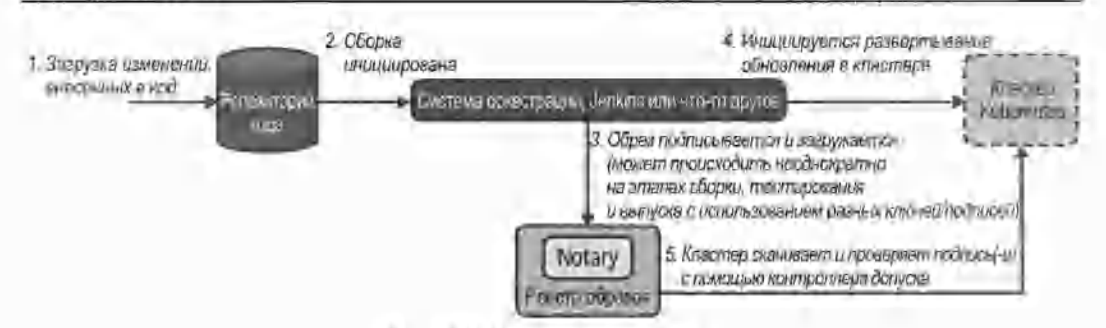


Рис. 15.3. Процедура подписания

Во-вторых, Notary требует идентификации каждого образа с помощью глобально уникальных имен GUN (Globally Unique Name), в состав которых входит URL-адрес реестра. Это затрудняет подписание, если реестров больше одного (например, один для кэша, другой для пограничных серверов и т. д.), так как подписи привязываются к отдельно взятому реестру, и их нельзя переназначать или копировать.

Наконец, Notary и TUF требуют, чтобы в процессе подписания применялись разные пары ключей. Каждый ключ имеет свои требования к безопасности, и в случае проникновения в системы его замена может быть проблематичной. В академическом смысле, текущая реализация Notary/TUF является хорошо продуманным решением, но она имеет слишком высокий порог вхождения для организаций, которые только начинают чувствовать себя уверенно при использовании некоторых базовых технологий. В связи с этим многие оказались не готовы променять удобство и уже имеющиеся знания на дополнительные преимущества с точки зрения безопасности, которые предлагает процедура подписания.

На момент написания этих строк существует инициатива по разработке и выпуску второй версии Notary. Эта обновленная версия должна быть более удобной за счет решения многих из тех проблем, которые мы только что обсуждали, включая упрощение управления ключами и поддержку их передачи за счет упаковывания их вместе с самими образами OCI.

Существуют несколько проектов, реализующих веб-хук допуска, который проверяет образы и, прежде чем пропускать их в кластер Kubernetes, убеждается в том, что они были подписаны. Мы ожидаем, что, как только недостатки этого подхода булут устранены, его начнут чаще внедрять в цепочку доставки ПО, а эти веб-хуки допуска, проверяющие подписи, станут еще более зрелыми.

Непрерывная доставка

В предыдущих разделах мы подробно обсудили процесс преобразования исходного кода в образ контейнера. Мы также поговорили о том, где эти образы хранятся и какие архитектурные и процессные решения необходимо принять при выборе и развертывании реестров образов. В этом заключительном разделе мы исследуем весь конвейер, который объединяет эти подготовительные этапы с самим развертыванием образов в одном или нескольких кластерах Kubernetes с разными

окружениями (предназначенными для тестирования, финального тестирования и эксплуатации).

Вначале мы покажем, как интегрировать процесс сборки в автоматический конвейер, а затем рассмотрим императивные — управляемые через проталкивание изменений (push-driven) конвейеры, с которыми многие уже знакомы. В конце будут представлены некоторые принципы и инструменты, появившиеся в области GitOps, представляющей собой относительно новый подход к развертыванию, который использует репозитории систем управления версиями в качестве достоверного источника ресурсов, предназначенных для размещения в наших окружениях.

Стоит отметить, что непрерывная доставка — это обширная область, которой посвящено много книг. В этом разделе мы исходим из того, что вы уже имеете общее представление о конвейерах CD, и уделяем основное внимание реализации этих принципов в рамках Kubernetes и сопутствующего инструментария.

Интеграция процесса сборки в конвейер

Когда разработчики пишут и тестируют код у себя на компьютере, они могут собирать образы локально с помощью Docker. Однако на всех последующих этапах процесс сборки должен быть частью автоматического конвейера, который запускается в ответ на фиксацию кода в репозитории центральной системы управления версиями. Позже в этой главе мы затронем более развитые методы, имеющие непосредственное отношение к развертыванию образов в окружении, но в данном разделе нас интересует исключительно выполнение этапов сборки в кластере с использованием облачно-ориентированных средств автоматизации.

Нам, как правило, хочется, чтобы процесс сборки инициировался в ответ на фиксацию кода. Некоторые инструменты, входящие в состав конвейера, периодически проверяют набор репозиториев, указанных в конфигурации, и при обнаружении изменений запускают задание. Тот же процесс может быть запущен путем вызова веб-хука из системы управления версиями. Чтобы проиллюстрировать некоторые идеи, представленные в этом разделе, мы позаимствуем несколько примеров из Tekton, популярного средства создания конвейеров, предназначенного для выполнения в Kubernetes. Tekton (и многие другие инструменты, ориентированные на Kubernetes) описывают компоненты конвейера с помощью CRD. В листинге 15.2 мы видим (сильно видоизмененный) экземпляр пользовательского ресурса `Task`, который можно многократно применять в разных конвейерах. Tekton предлагает каталог распространенных действий (например, в следующем фрагменте кода это клонирование git-репозитория), которые мы можем задействовать в наших собственных конвейерах.

Листинг 15.2

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: git-clone
```

```
spec:
  workspaces:
    - name: output
      description: "The git repo will be cloned onto the \
      volume backing this workspace"
  params:
    - name: url
      description: git url to clone
      type: string
    - name: revision
      description: git revision to checkout (branch, tag, sha, ref…)
      type: string
      default: master
    <...опущено...>
  results:
    - name: commit
      description: The precise commit SHA that was fetched by this Task
  steps:
    - name: clone
      image: "gcr.io/tekton-releases/github.com/tektoncd/\
      pipeline/cmd/git-init:v0.12.1"
      script: |
        CHECKOUT_DIR="$(workspaces.output.path)/$(params.subdirectory)"
        <...опущено...>
        /ko-app/git-init \
          -url "$(params.url)" \
          -revision "$(params.revision)" \
          -refspec "$(params.refspec)" \
          -path "$CHECKOUT_DIR" \
          -sslVerify="$(params.sslVerify)" \
          -submodules="$(params.submodules)" \
          -depth "$(params.depth)"
        cd "$CHECKOUT_DIR"
        RESULT_SHA="$(git rev-parse HEAD | tr -d '\n')"
        EXIT_CODE="$?"
        if [ "$EXIT_CODE" != 0 ]
        then
          exit $EXIT_CODE
        fi
        # Убедитесь, что отсутствует завершающий символ новой строки в
результате.!
        echo -n "$RESULT_SHA" > $(results.commit.path)
```

Как уже упоминалось в предыдущих разделах, образы OCI можно создавать множеством разных способов. Некоторые из них требуют наличия файла Dockerfile, а другие — нет. Также в ходе сборки может понадобиться выполнение дополнительных действий. Почти во всех средствах построения конвейеров применяются такие

понятия, как стадии, этапы или задания, с помощью которых пользователи могут настраивать отдельные наборы возможностей и объединять их в цепочки. В листинге 15.3 приведен пример определения `Task`, которое собирает образ с помощью Cloud Native Buildpacks.

Листинг 15.3

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: buildpacks-phases
  labels:
    app.kubernetes.io/version: "0.1"
  annotations:
    tekton.dev/pipelines.minVersion: "0.12.1"
    tekton.dev/tags: image-build
    tekton.dev/displayName: "buildpacks-phases"
spec:
  params:
    - name: BUILDER_IMAGE
      description: "The image on which builds will run \
      (must include lifecycle and compatible buildpacks)."
    - name: PLATFORM_DIR
      description: The name of the platform directory.
      default: empty-dir
    - name: SOURCE_SUBPATH
      description: "A subpath within the `source` input \
      where the source to build is located."
      default: ""

  resources:
    outputs:
      - name: image
        type: image

  workspaces:
    - name: source

  steps:
    <...опущено...>
    - name: build
      image: $(params.BUILDER_IMAGE)
      imagePullPolicy: Always
      command: ["/cnb/lifecycle/builder"]
      args:
        - "-app=$(workspaces.source.path)/$(params.SOURCE_SUBPATH)"
        - "-layers=/layers"
```

```
      - "-group=/layers/group.toml"
      - "-plan=/layers/plan.toml"
    volumeMounts:
      - name: layers-dir
        mountPath: /layers
      - name: $(params.PLATFORM_DIR)
        mountPath: /platform
      - name: empty-dir
        mountPath: /tekton/home
  <...опущено...>
```

Затем мы можем привязать это и другие задания к нашему входному репозиторию в рамках конвейера Pipeline (который здесь не показан). Для этого нужно связать между собой два рабочих пространства: то, в которое мы ранее клонировали наш git-репозиторий, и то, которое наш конструктор пакетов сборки будет использовать в качестве источника. Мы также можем указать, что по завершении процесса образ должен быть загружен в реестр.

Благодаря гибкости рассмотренного подхода (а именно, настраиваемым блокам заданий) конвейеры становятся очень мощным средством описания процесса сборки в Kubernetes. У нас есть возможность добавить стадии тестирования и/или какого-нибудь статического анализа кода. Мы также могли бы при желании определить в нашем образе стадию подписания (см. раздел "Подписание образов" данной главы) или наше собственное задание для запуска других средств сборки таких, как Kaniko или BuildKit (если бы не использовались пакеты сборки, как в этом примере).

Развертывание на основе загрузки

В предыдущем разделе вы узнали, как автоматизируется сборка в конвейере. Здесь же мы покажем, как расширить этот процесс, чтобы в его рамках выполнялось развертывание в кластере. Мы также обсудим несколько методов, которые помогут упростить такого рода автоматическую доставку.

Благодаря гибкости подхода на основе заданий/стадий, который был представлен ранее (и поддерживается почти любым инструментом), нам не составит труда создать в конце конвейера этап, на котором происходит чтение тега только что созданного (и загруженного) образа и обновления соответствующего объекта `Deployment`. Это можно было бы сделать путем обновления `Deployment` непосредственно в кластере с помощью команды `kubectl set image`, и этот подход до сих пор демонстрируется в нескольких статьях и практических руководствах. Но есть более удачная альтернатива: мы можем сделать так, чтобы наш конвейер записал измененный тег образа обратно в YAML-файл с описанием объекта `Deployment` и затем зафиксировал эти изменения в системе управления версиями. После этого можно выполнить команду `kubectl apply` с указанием новой версии репозитория, чтобы изменения вступили в силу. Это более предпочтительный подход, так как мы можем и дальше использовать YAML-файл в качестве достоверного источника данных для нашего кластера (подробней об этом в разделе "GitOps" данной главы), но первый вариант

тоже годится в качестве промежуточного этапа при переходе на такого рода автоматический конвейер, ориентированный на Kubernetes.

Метаданные образа и назначение ему тегов

Нам периодически приходится обсуждать с нашими клиентами тему выбора названий и/или версий для образов, и мы видели некоторые распространенные методы, которые работают в большинстве случаев. Когда образы собираются сразу после редактирования кода, в качестве тега хорошо подходит хеш git. Для человека подобные хеши выглядят бессмысленными, но, если у вас есть автоматические конвейеры, это обычно не проблема. Они являются удобной ссылкой на фиксацию кода.

По мере развития конвейера рекомендуется переходить к более наглядным тегам с версиями (например, `SemVer`), чтобы вам с первого взгляда было легко понять, какие версии развернуты в ваших окружениях. Также попробуйте использовать неизменяемые теги (которые поддерживаются в большинстве реестров), чтобы они не перезаписывались. В результате вы сможете быть уверены в том, что скачиваемые вами образы с одним и тем же тегом будут одинаковыми.

Крайне полезным также может быть добавление в образы метаданных (меток) с контактной информацией о владельце и, возможно, с описанием сборки. Эти метаданные могут использоваться в отчетах на более поздних этапах или в других инструментах, таких как диспетчеры политик, которые могут предоставлять какой-то уровень доступа к образу на основе определенных меток.

При развертывании приложений в Kubernetes мы имеем дело с двумя видами артефактов: кодом и конфигурацией, необходимой приложению и описывающей то, как оно *собирается* и *развертывается*. У нас часто спрашивают мнение о том, как эти артефакты лучше всего организовать. Кто-то предпочитает хранить *все*, что относится к приложению, вместе в единой иерархии, а кому-то удобней эти вещи разделять.

Мы обычно советуем второй подход, для этого есть несколько причин:

- Эти артефакты обычно относятся к разным предметным областям или группам в организации. Разработчики должны знать, как развертываются их приложения, и у них должно быть какое-то влияние на этот процесс, однако за конфигурацию, связанную с размерами кластеров, окружениями, внедрением конфиденциальных данных и т. д., в основном отвечают администраторы платформы или эксплуатационная команда.
- Репозитории с кодом и с такими ресурсами, как артефакты конвейера развертывания, конфиденциальные данные и конфигурация окружения, скорее всего, имеют разные требования к привилегиям и аудиту безопасности.

При размещении своей конфигурацией развертывания в отдельном репозитории, весь процесс упрощается: сначала мы скачиваем этот репозиторий, затем обновляем тег образа (с помощью `sed` или аналогичной утилиты) и в конце *загружаем изменения обратно в git*, чтобы наш репозиторий оставался достоверным источником данных. После этого для измененных манифестов можно выполнить команду `kubectl apply -f`.

Эта императивная модель (*на основе загрузки*) обеспечивает отличную проверяемость, так как нам доступны средства журналирования и создания отчетов, встроенные в систему управления версиями, и можно без труда наблюдать за продвижением изменений по нашему конвейеру, как показано на рис. 15.4.



Рис. 15.4. Развертывание на основе загрузки

В зависимости от того, насколько автоматизированы процессы в вашей организации, вы можете сделать так, чтобы ваш конвейер включал в себя разные окружения, или даже чтобы развертывания происходили в разных кластерах Kubernetes. Конечно, это можно реализовать с помощью большинства инструментов, и некоторые из них лучше поддерживают такие операции. Однако здесь именно тот случай, когда воплощение модели императивных конвейеров, описанной в данном разделе, может быть затруднительной, поскольку нам нужно хранить ресурсы (и учетную информацию) для всех кластеров, которые мы хотим использовать для развертывания.

Еще один недостаток этого подхода (когда одна центральная система загружает изменения в наши окружения) в том, что, если работа нашего конвейера по какой-то причине будет нарушена, нам придется его перезапустить или вернуть обратно в рабочее состояние. Мы также должны организовать мониторинг и систему оповещений для наших конвейеров (независимо от их реализации), чтобы всегда знать о любых проблемах, возникающих с нашими развертываниями.

Методы выкатывания изменений

В конце предыдущего раздела мы упомянули о том, что для успешного завершения работы конвейеров требуется мониторинг. Однако при развертывании новых версий приложения у нас также должна быть возможность отслеживать их работоспособность, чтобы понимать, стоит заниматься исправлением проблем или лучше откатиться до предыдущего рабочего состояния.

Организациям доступно на выбор несколько методик. Им посвящены целые книги, но здесь мы проведем их краткий обзор, чтобы показать, как они могут быть реализованы в Kubernetes:

- *Канареечные развертывания* — новая версия приложения в кластере, и к ней направляется небольшая часть трафика (в зависимости от метаданных, пользователей или каких-то других атрибутов). За ней можно пристально следить, чтобы убедиться в том, что она ведет себя так же, как предыдущий выпуск, или, по крайней мере, не приводит к ошибкам. По мере того, как растет наша уверенность в работоспособности новой версии, мы можем постепенно увеличивать долю трафика, которая на нее приходится.
- *Сине-зеленые развертывания* — подход похож на канареечные развертывания, но подразумевает более резкое распределение трафика. Для этого подойдет как один, так и несколько кластеров (один со старой, *синей* версией, и еще один с новой, *зеленой*, версией). Суть в том, что, прежде чем направлять весь трафик к новой версии, мы можем подтвердить, что развертывание сервиса проходит без неожиданностей, и выполнить тестирование окружения, с которым не взаимодействуют конечные пользователи. Если наблюдается повышенное количество ошибок, мы можем вернуть трафик обратно. У этого подхода, естественно, есть дополнительные нюансы, так как вам, возможно, придется реализовать в своих приложениях безопасную работу с состоянием, сеансами и другими вещами.
- *A/B-тестирование* — подобно канареечным развертываниям, мы выкатываем какой-то части наших потребителей версию приложения с потенциально измененным поведением. Мы можем собирать метрики и анализировать закономерности функционирования этой новой версии, чтобы решить, откатывать ее назад, выкатывать полностью или расширить рамки эксперимента.

Рассмотренные методы приближают нас к нашей цели, состоящей в разделении развертывания приложений и их выпуска для конечных потребителей. Это позволяет нам выбрать подходящий момент для включения определенных возможностей и/или перехода на новую версию. Данные подходы помогают сделать развертывание изменений в наших окружениях куда менее рискованным.

Большинство этих методов реализуются в виде какой-то системы для переключения трафика. В Kubernetes это возможно благодаря очень функциональным механизмам и богатым возможностям для работы с сетью. Один из открытых инструментов с поддержкой этих подходов (а также различных решений для организации mesh-сетей) — Flagger — работает внутри Kubernetes в виде контроллера и следит за изменениями поля `image` в ресурсах `Deployment`. Он предоставляет множество параметров, которые делают возможной поддержку перечисленных методов за счет автоматизированного изменения конфигурации исходной mesh-сети для переключения трафика нужным нам образом. Кроме того, Flagger позволяет следить за работоспособностью недавно развернутых версий, чтобы либо продолжить процесс их развертывания, либо остановить его с последующим откатом изменений.

Мы считаем, что Flagger и другие решения в этой области определенно заслуживают внимания. Но, по нашему опыту, эти подходы обычно начинают обсуждаться

только на второй или третьей стадии внедрения Kubernetes ввиду дополнительных компонентов, от которых они зависят (для реализации большинства методов требуется mesh-сеть).

GitOps

До сих пор мы рассматривали процесс добавления стадии развертывания в конвейеры доставки для Kubernetes. В настоящее время в этой области набирает популярность альтернативная модель под названием GitOps. Вместо императивной загрузки изменений в кластер GitOps предлагает контроллер, который постоянно согласовывает содержимое git-репозитория с ресурсами кластера, как показано на рис. 15.5. Такая модель во многом напоминает работу управляющего цикла согласования, который нам предоставляет сама платформа Kubernetes. Двумя основными инструментами в сфере GitOps являются ArgoCD и Flux, их авторы совместно работают над созданием общего ядра, которое должно стать основой обоих проектов.



Рис. 15.5. Принцип работы GitOps

У этой модели есть несколько важных достоинств:

- Она декларативна по своему характеру, поэтому в случае возникновения какихлибо проблем с развернутой версией ее ручного удаления система попытается прийти к корректному состоянию.
- Git становится единым достоверным источником данных, и мы можем задействовать уже имеющиеся навыки и опыт работы с этим инструментарием, не говоря уже о том, что мы без всяких усилий получаем надежный журнал аудита изменений. Мы можем использовать процесс создания запросов на включение внесенных изменений в качестве интерфейса для обновления кластеров и по мере необходимости интегрировать существующие инструменты с помощью механизмов расширения, которые предоставляются большинством систем управления версиями (таких как веб-хуки, рабочие процессы, действия и т. д.).

Однако описанный подход не свободен от недостатков. Если организация действительно желает использовать git в качестве единого источника достоверной информации, ей придется хранить конфиденциальные данные в системе управления версиями. За последние годы появилось несколько проектов, призванных решить эту проблему, самым известным из которых является Sealed Secrets от Bitnami. Этот проект позволяет фиксировать в репозитории зашифрованные версии объектов `Secret` и затем расшифровывать их, когда они применяются к кластеру (чтобы они были доступны для приложений). В *главе 5* этот подход обсуждается более подробно.

Нам также необходимо следить за корректностью процесса синхронизации. Если конвейеру, основанному на загрузке, не удастся выполнить свою работу, мы это увидим. Но ввиду декларативного характера GitOps нам нужно организовать оповещения на случай, если наблюдаемое (в кластере) и объявленное (в git) состояния остаются несогласованными на протяжении длительного времени.

В нашей работе мы наблюдаем активное внедрение GitOps, хотя это, несомненно, совершенно новая парадигма по сравнению с моделями на основе загрузки. Не все приложения можно развернуть без проблем путем произвольного применения YAML-ресурсов. Изначально в процессе перехода на GitOps вам, возможно, придется создать механизм упорядочивания и предусмотреть какое-то скриптование.

Также необходимо знать, какие инструменты могут создавать, изменять или удалять ресурсы в рамках своего жизненного цикла, так как их иногда необходимо адаптировать к модели GitOps. В качестве примера можно привести контроллер, который работает в кластере, следит за определенным CRD и затем создает другие ресурсы непосредственно с помощью API-интерфейса Kubernetes. В строгом режиме инструменты GitOps могут удалять эти динамически создаваемые ресурсы, так как они не описаны в едином источнике достоверных данных (репозитории git). Конечно, в большинстве случаев такое удаление неизвестных ресурсов является желаемым и иллюстрирует одно из положительных качеств GitOps. Но вы должны иметь четкое представление о ситуациях, в которых изменения могут намеренно вноситься в обход репозитория git, нарушая работу этой модели. Для них нужно предусмотреть какое-то отдельное решение.

Резюме

В этой главе мы рассмотрели процесс упаковывания исходного кода в контейнер с последующим развертыванием его в кластере Kubernetes. Многие этапы и принципы, с которыми вы уже знакомы (сборка/тестирование, CI, CD и т. д.), в той же степени применимы и к окружениям на основе контейнеров/Kubernetes, только с использованием других инструментов. Хотя некоторые идеи (такие как GitOps) могут быть для вас новыми. Они являются развитием концепций, которые поддерживаются самой платформой Kubernetes, и направлены на повышение надежности и безопасности в рамках существующих моделей развертывания.

В этой области существует множество инструментов, позволяющих реализовать много разных процессов и подходов. Однако один из ключевых выводов, к которым вы должны были прийти в этой главе, состоит в том, что вам необходимо решить, какой доступ к каждой части конвейера должна иметь та или иная группа в организации. Возможно, разработчики активно применяют Kubernetes и достаточно изолированы для того, чтобы самостоятельно создавать артефакты для сборки и развертывания (или, по крайней мере, играть важную роль в этом процессе). А может, вам хочется скрыть от них все внутренние детали, чтобы облегчить масштабирование и стандартизацию, и при этом возложить дополнительные обязанности по внедрению необходимых основополагающих механизмов и средств автоматизации на администраторов платформы.

ГЛАВА 16

Абстрагирование платформы

Мы неоднократно видели, как организации проектировали и реализовывали платформы Kubernetes по принципу *главное сделать, а пользователи сами собой появятся*. Однако во многих случаях эта философия чревата тем, что в платформе могут быть не учтены ключевые требования специалистов, которые будут с ней взаимодействовать (таких как разработчики, отдел информационной безопасности, сетевые администраторы и т. д.). В результате многое приходится переделывать, что требует дополнительных усилий. Чтобы итоговая платформа соответствовала своему назначению, в ее создании должны участвовать разные группы.

В этой главе мы обсудим некоторые факторы, которые необходимо учитывать при проектировании процесса перехода других команд (в частности, разработчиков) на вашу платформу Kubernetes. Вначале будут рассмотрены общие аспекты и вопросы. Насколько глубоко разработчики должны знать Kubernetes? Затем мы перейдем к обсуждению того, как сделать так, чтобы разработчики могли без особых трудностей приступить к размещению своего кода в Kubernetes и развертыванию самих кластеров. Наконец, мы вернемся к спектру сложности, который упоминался в *главе 1*, и поговорим о том, какие уровни абстрагирования мы можем внедрить. Наша цель состоит в том, чтобы сделать платформу Kubernetes подходящей для разработчиков с разным уровнем знаний и желанием взаимодействовать с исходной реализацией. Для этого нужно найти хороший баланс между сложностью и гибкостью.

Многие темы, обсуждаемые в данной главе, уже рассматривались ранее, и при необходимости мы будем об этом упоминать. Здесь мы хотим взглянуть на эти аспекты в контексте улучшения взаимодействия между командами и создания платформы, которая отвечает нуждам всех, кто работает в организации. И хотя на первый взгляд данная тема может показаться легкой, обсуждаемые здесь проблемы становятся серьезнейшим камнем преткновения для многих компаний, и от того, удастся ли их преодолеть, зависит успех внедрения платформы приложений на основе Kubernetes.

Открытость платформы

В этой книге мы уже неоднократно говорили о том, что в процессе проектирования и реализации платформы Kubernetes вы должны анализировать свои конкретные требования и задавать вопросы, относящиеся к разным областям. Один из главных вопросов, который ляжет в основу многих ваших решений, заключается в том, насколько открытыми должны быть внутренние системы и ресурсы Kubernetes для групп разработки. Ответ на него зависит от нескольких факторов.

Kubernetes — относительно новая технология. В некоторых случаях стремление к ее внедрению исходит от инфраструктурной части организации и продиктовано желанием стандартизировать приложения или сделать использование платформы более простым и эффективным. Бывает и так, что разработчики сами инициируют внедрение Kubernetes, движимые желанием реализовать новую технологию, которая, как им кажется, может улучшить и ускорить процессы создания и развертывания облачно-ориентированных приложений. Откуда бы ни исходила инициатива, последствия внедрения ощутят на себе и другие команды; это может быть связано с принятием новой парадигмы, изучением новых инструментов или просто тем фактом, что с новой платформой придется взаимодействовать по-новому.

В некоторых организациях существует жесткое требование относительно того, что команды разработки не должны иметь дело с исходной платформой. В основе этого лежит мнение о том, что разработчики должны сосредоточиться на удовлетворении бизнес-требований и не отвлекаться на детали реализации разрабатываемой платформы. Подобный подход не лишен смысла, но на практике мы не всегда с ним согласны. Например, чтобы сделать процесс создания приложений для платформы эффективным, у разработчиков должно быть хоть *какое-то* представление о том, как она устроена. Это не означает, что приложения необходимо сильнее привязывать к платформе — просто нужно понимать, как лучше всего пользоваться ее возможностями. Более подробно об этих отношениях между платформой и приложениями можно почитать в *главе 14*.

Успех модели, в которой внутреннее устройство платформы *недоступны для разработчиков*, зависит от того, достаточно ли ресурсов у команды, которая занимается развитием платформы. Во-первых, так как на нее ложится вся ответственность за сопровождение и поддержку окружений, и, во-вторых, потому что эта же команда будет отвечать за создание всех абстракций, необходимых разработчикам для беспроблемного взаимодействия с платформой. Это важно, поскольку, даже если разработчики не взаимодействуют с Kubernetes напрямую, им все равно нужно как-то анализировать производительность приложений, отлаживать ошибки и проводить диагностику. Если предоставление доступа к утилите `kubectl` раскрывает слишком много внутренних аспектов кластера, необходимо предусмотреть промежуточный слой, который позволит разработчикам *полноценно администрировать* свои приложения в реальной среде, но при этом не будет нагружать их деталями реализации. В *главе 9* мы обсуждаем многие из основных способов эффективного предоставления средств отладки командам разработки.

В некоторых организациях одного лишь повышения удобства использования разработчиками средств диагностики может быть недостаточно. Развертывание приложений в Kubernetes тоже может оказаться сложным процессом, требующим наличия множества компонентов. Возможно, для успешного развертывания приложения потребуются такие ресурсы, как `StatefulSet`, `PersistentVolumeClaim`, `Service` и `ConfigMap`. Если предоставление этих ресурсов команде разработки нежелательно, вы можете пойти еще дальше и создать вокруг них абстракции. Для этого можно применить конвейеры самообслуживания или создать пользовательские ресурсы

(это обсуждается в *главе 11*), чтобы упростить инкапсуляцию необходимых элементов. Оба эти подхода будут рассмотрены позже в этой главе.

Сдерживающим фактором в принятии решения о том, насколько открытой должна быть платформа, являются умения и опыт тех, кто создает эти абстракции. Например, командам, которые занимаются развитием платформы, нужны определенные навыки программирования и знание рекомендованных подходов к использованию API-интерфейса Kubernetes, иначе они не смогут пойти по пути создания пользовательских ресурсов. Если такие умения и знания отсутствуют, спектр абстракций, который вы сможете предложить, может быть довольно узким, в результате чего вам придется открыть командам разработки более широкий доступ к внутреннему устройству платформы.

В следующем разделе будут освещены некоторые способы предоставления разработчикам (и другим конечным пользователям) модели самообслуживания с целью облегчения и стандартизации развертывания как приложений, так и кластеров.

Самостоятельное присоединение к платформе

На начальных этапах перехода на Kubernetes за создание и конфигурацию кластеров для всех желающих, скорее всего, будет отвечать команда развития платформы. Она также, вероятно, должна будет, как минимум, помогать с развертыванием приложений в этих кластерах. Требования к этому подготовительному процессу будут зависеть от выбранной вами модели использования кластера (подробности о моделях использования ресурсов для приложений можно найти в *главе 12*). В однотенантной модели создание и конфигурация кластеров может быть более простой, основанной на наборе общих прав доступа, основных сервисах (журналирование, мониторинг, управление входящим трафиком и т. д.) и настройке доступа (технология единого входа). Однако в мультитенантном кластере нам, возможно, придется создать множество дополнительных компонентов (таких, как пространства имен, сетевые политики и квоты) для каждой команды и приложения, которые присоединяются к платформе.

Однако по мере того как организация начинает расширяться, ручное выделение и конфигурация ресурсов становится непосильной задачей. Это вынуждает администраторов платформы постоянно выполнять одну и ту же работу, а командам разработки тем временем приходится ждать. Команды, достигшие базового уровня зрелости, обычно начинают предлагать своим внутренним пользователям какого-то рода системы самообслуживания. Чтобы делать это эффективно, можно воспользоваться имеющимся средством CI/CD или таким процессом, как Jenkins или GitLab. Оба эти инструмента позволяют легко создавать конвейеры и дают возможность передавать дополнительный видоизмененный ввод на этапе выполнения.

Благодаря зрелости таких инструментов, как kubeadm и Cluster API процесс автоматизации создания кластера является относительно простым и предсказуемым. Команды могут предоставлять настраиваемые параметры такие как имя или размер кластера, а конвейер в свою очередь может использовать эти инструменты для соз-

дания кластеров с разумными параметрами по умолчанию, прежде чем предоставлять запрашивающей команде подходящие учетные данные или права доступа. Как это часто бывает, сложность вашей автоматизации зависит лишь от вас. Нам встречались конвейеры, которые автоматически создают балансировщики нагрузки и DNS-серверы в зависимости от сведений о запрашивающем пользователе, полученных из LDAP, включая автоматическую маркировку исходной инфраструктуры с указанием центров издержек. В зависимости от команды, окружения или проекта, пользователь может иметь возможность выбирать размер кластера, но в определенных рамках. Мы даже можем позволить выбирать между открытым и приватным облаком, учитывая категорию или профиль безопасности приложения. У администраторов платформы есть широкий спектр возможностей создания гибкого, но в то же время мощного автоматического процесса выделения ресурсов для команд разработки.

В условиях мультитенантности вместо кластеров создаются пространства имен со всеми сопутствующими объектами, позволяющими предоставить новому приложению среду с нестрогой изоляцией. Опять же, мы можем применить аналогичный подход на основе конвейеров, но при этом позволить командам разработки выбирать кластер (или кластеры), в которых будет развернуто их приложение. Нам нужно сгенерировать как минимум следующее:

- *Пространство имен* — чтобы разместить в нем приложение и обеспечить логическую изоляцию, на которую могут опираться другие наши компоненты.
- *RBAC* — чтобы только как следует авторизованные группы могли иметь доступ к ресурсам в пространстве имен своего собственного приложения.
- *Сетевые политики* — чтобы приложению было позволено взаимодействовать только с самим собой или другими разделяемыми сервисами кластера, но *не* с другими приложениями в том же кластере (если есть такое требование).
- *Квоты* — чтобы ограничить количество ресурсов, которые может потребить одно пространство имен или приложение и снизить тем самым риск возникновения проблемы *шумного соседа*.
- *Ограничительные диапазоны* — чтобы установить разумные значения по умолчанию для определенных объектов, созданных в пространстве имен.
- *Политики безопасности Pod'ов* — чтобы приложения соблюдали параметры безопасности, установленные по умолчанию, такие, как запрет на запуск от имени администратора (root).

В зависимости от ситуации некоторые из этих механизмов могут не потребоваться, хотя вместе они позволяют администраторам кластера и платформы наладить беспроблемный процесс присоединения и среду развертывания для новых команд разработки, не требуя ручного вмешательства.

Когда организация начнет лучше ориентироваться в Kubernetes, эти конвейеры можно будет реализовать стандартными средствами Kubernetes с использованием операторов. Например, мы можем определить ресурс `Team` так, как показано в листинге 16.1.

Листинг 16.1

```
apiVersion: examples.namespace-operator.io/v1
kind: Team
metadata:
  name: team-a
spec:
  owner: Alice
  resourceQuotas:
    pods: "50"
    storage: "300Gi"
```

В этом примере можно определить конкретную команду, которую мы хотим присоединить к платформе, с владельцем (пользователем) и какими-то квотами на ресурсы. Наш контроллер, размещенный в кластере, будет отвечать за чтение этого объекта, а также за создание и интеграцию соответствующего пространства имен, ресурсов RBAC и квот. Этот подход может быть эффективным, так как он позволяет активно задействовать API-интерфейс Kubernetes и предлагать стандартные средства администрирования и согласования ресурсов. Например, при случайном удалении роли или изменении квоты контроллер сможет автоматически исправить ситуацию. Ресурсы более высокого уровня (такие, как `Team` или `Application`) тоже могут отлично подойти для начальной конфигурации кластера, но, добавив несколько объектов `Team` в сочетании с нашим контроллером, мы можем автоматизировать всю необходимую конфигурацию и подготовить ее к использованию.

Мы определенно можем пойти еще дальше и создать сложную систему автоматизации. Подумайте, к примеру, о средствах наблюдаемости, которые могут понадобиться новым приложениям. Мы также можем сделать так, чтобы наш контроллер `Team` генерировал и предоставлял новым командам или приложениям информационные панели, адаптированные специально для них, и чтобы система Grafana их автоматически перезагружала. Мы можем динамически добавлять новые цели оповещений в Alertmanager для присоединяющихся команд или пространств имен. За этими простыми и дружественными к пользователям абстракциями могут скрываться богатые возможности.

Спектр абстрагирования

В *главе 1* было представлено понятие *спектра абстрагирования*. На рис. 16.1 мы его расширили и добавили несколько конкретных уровней.

В предыдущих разделах обсуждались некоторые философские аспекты и организационные ограничения, которые могут повлиять на то, в каком месте этого спектра окажетесь вы. Здесь же мы пройдемся более подробно по этому спектру слева направо (от платформы без абстракций к полностью абстрактной платформе) и заодно обсудим некоторые варианты и компромиссы.



Рис. 16.1. Спектр абстрагирования

Инструменты командной строки

Предоставляя доступ к API-интерфейсу Kubernetes посредством стандартных инструментов командной строки, мы попадаем в крайний левый конец спектра, который исключает какие-либо абстракции. В некоторых организациях kubectl служит основным средством взаимодействия разработчиков с Kubernetes. Причиной этого могут быть как ограничения (отсутствие поддержки со стороны администраторов платформы), так и сознательный выбор (например, если разработчики уже знакомы с Kubernetes и желают работать с этой системой напрямую). При этом в кластере по-прежнему могут быть предусмотрены средства автоматизации или ограничительные меры, но разработчики будут взаимодействовать с ним с помощью стандартных инструментов.

У этого подхода есть несколько недостатков (даже если ваши команды разработки *действительно* имеют какое-то представление о Kubernetes):

- Необходимость ручной подготовки и настройки методов аутентификации для потенциально большого числа кластеров может быть обременительной. Это относится и к переключению между контекстами разных кластеров и гарантии того, что пользователи всегда работают с нужным им кластером.
- Формат вывода команд kubectl может быть неудобным для просмотра и анализа. По умолчанию вывод имеет вид таблицы, но его можно преобразовывать в другие форматы и передавать таким инструментам, как jq для более компактного представления информации. Но для этого разработчики должны быть знакомы с параметрами kubectl (а также внешних инструментов) и уметь с ними работать.
- За счет прямого доступа к kubectl пользователи получают в свое распоряжение все возможности Kubernetes без каких-либо абстрактных или промежуточных слоев. В связи с этим нам нужно не только предусмотреть подходящие правила RBAC, чтобы ограничить неавторизованный доступ, но также слой управления допуском, который проверяет все запросы к API-серверу.

Этот подход можно сделать более удобным с помощью различных инструментов. У kubectl есть много подключаемых модулей, которые могут повысить удобство работы с локальной командной оболочкой. Например, модули kubens и kubectx облегчают использование пространств имен и, соответственно, контекстов, делая и те,

и другие более прозрачными. Есть подключаемые модули, которые агрегируют журнальные записи из разных Pod'ов или выводят информацию о работоспособности приложения с помощью консольного пользовательского интерфейса. Это не самые передовые инструменты, но они позволяют избавиться от распространенных проблемных моментов и сделать так, чтобы разработчикам больше не нужно было вникать в тонкости некоторых внутренних аспектов реализации. Однако, несмотря на наличие этих полезных вспомогательных компонентов, мы по-прежнему предоставляем прямой доступ к API-интерфейсу Kubernetes, почти никак его не абстрагируя.

Существуют также подключаемые модули, которые интегрируются с внешними системами, чтобы упростить процесс аутентификации и скрыть от пользователя такие сложные механизмы, как `kubeconfig`, сертификаты, токены и т.д. В этой области мы регулярно наблюдаем попытки расширить стандартный инструментарий, так как предоставление разработчикам безопасного доступа к нескольким кластерам (особенно тем, которые могут создаваться и удаляться динамически) может быть непростой задачей. Во второстепенных окружениях доступ может быть основан на парах ключей (которые нужно генерировать, администрировать и распространять). Если же речь идет о более стабильном окружении, то для получения доступа в большинстве случаев применяется система единого входа. Мы разработали для нескольких наших клиентов утилиты командной строки, которые извлекают учетные данные из центрального реестра кластера, используя ту информацию, которую пользователь ввел при входе в систему.

Кроме того, вы можете пойти по пути компании Airbnb. В ходе недавней презентации на конференции QCon (**https://oreil.ly/OxTSc**) Мелани Себула рассказала, как в Airbnb разрабатывают более развитые наборы инструментов (в виде отдельных исполняемых файлов и подключаемых модулей `kubectl`) для взаимодействия со своими кластерами, а также вмешиваются с помощью хуков в процесс сборки, развертывания и прочих операций с образами.

Существует еще одна категория инструментов, позволяющая разработчикам взаимодействовать с кластером посредством графического интерфейса. Среди новых популярных решений в этой области можно выделить Octant (**https://octant.dev**) и Lens (**https://k8slens.dev**). Эти инструменты работают не внутри кластера, как информационная панель Kubernetes, а локально, на рабочем компьютере, и используют для доступа к кластеру `kubeconfig`. Они могут сильно помочь разработчикам, которые плохо знакомы с платформой и хотят получить визуальное представление кластера и своих приложений. Повышение удобства работы на стороне клиента может послужить тем шагом, с которого организация начнет упрощать взаимодействие разработчиков с Kubernetes.

Абстрагирование посредством шаблонизации

Развертывание одного приложения в Kubernetes может потребовать создания множества объектов. Например, *одному* приложению Wordpress может понадобиться следующее:

- *Deployment* — описание образов, команд и свойств экземпляра Wordpress.

- *StatefulSet* — развертывание MySQL в качестве хранилища данных для Wordpress.
- *Service* — предоставление механизмов обнаружения и балансировки нагрузки как для Wordpress, так и для MySQL.
- *PVC* — динамическое создание тома для данных Wordpress.
- *ConfigMap* — хранение конфигурации как для Wordpress, так и для MySQL.
- *Secret* — хранение администраторских учетных данных как для Wordpress, так и для MySQL.

В этом списке перечислено почти 10 разных объектов для поддержки одного крайне мелкого приложения. К тому же, для их настройки необходимы экспертные знания и понимание нюансов. Например, при использовании объекта `StatefulSet` нам нужно создать специальный *неуправляемый* Сервис, который будет принимать направленный к нему трафик. Мы хотим, чтобы наши разработчики могли развертывать свои приложения в кластере *без* необходимости знать, как самостоятельно создавать и настраивать все эти разные типы объектов Kubernetes.

Чтобы упростить развертывание этих приложений, мы можем предоставить разработчикам лишь небольшой набор параметров и генерировать все остальное автоматически. Такой подход не требует от разработчиков знания всех полей во всех объектах, но он по-прежнему раскрывает некоторые внутренние компоненты и использует лишь чуть более высокоуровневые средства по сравнению с обычной утилитой `kubectl`. Определенной зрелостью в этом отношении обладают инструменты шаблонизации, такие как Helm и Kustomize.

Helm

За последние несколько лет инструмент Helm приобрел популярность в экосистеме Kubernetes. Мы понимаем, что его возможности далеко не ограничиваются одной лишь шаблонизацией, но, по нашему опыту, описание шаблонов (с последующим редактированием и применением манифестов) является его более убедительной стороной в сравнении с его функциями управления жизненным циклом.

Далее показан фрагмент кода из чарта Helm для Wordpress (чарт — это пакет с описанием приложения), который описывает Сервис:

```
ports:
  - name: http
    port: {{ .Values.service.port }}
    targetPort: http
```

Данный шаблон не предоставляется разработчикам напрямую. Вместо этого он использует значение, которое внедряется или определяется в каком-то другом месте. В случае с Helm его можно передать с помощью командной строки или посредством файла значений (второй вариант более распространен):

```
## Конфигурация Kubernetes
## Для minikube; укажите NodePort, а в качестве типа - LoadBalancer или ClusterIP
##
service:
```

```
  type: LoadBalancer
  ## HTTP Port
  ##
  port: 80
```

Чарты содержат файл Values.yaml с разумными параметрами по умолчанию, но разработчики могут переопределить те из них, которые им нужны. Это позволяет легко видоизменять шаблоны даже тем, кто не обладает глубокими знаниями. Помимо обычной шаблонизации значений, Helm поддерживает элементарные логические операции, благодаря чему одно-единственное изменение в файле значений может сгенерировать или модифицировать большие участки соответствующих шаблонов.

Например, в представленном ранее файле значений есть объявление `type: Loadbalancer`. Оно напрямую внедряется в шаблон в нескольких местах, но при этом инициирует более сложную логику шаблонизации за счет наличия условных выражений и встроенных функций, как показано в листинге 16.2.

Листинг 16.2

```
spec:
  type: {{ .Values.service.type }}
  {{- if (or (eq .Values.service.type "LoadBalancer")
    (eq .Values.service.type "NodePort")) }}
  externalTrafficPolicy: {{ .Values.service.externalTrafficPolicy | quote }}
  {{- end }}
  {{- if (and (eq .Values.service.type "LoadBalancer")
    .Values.service.loadBalancerSourceRanges) }}
  loadBalancerSourceRanges:
  {{- with .Values.service.loadBalancerSourceRanges }}
{{ toYaml . | indent 4 }}
  {{- end }}
  {{- end }}
```

Подобная вложенная логика может показаться сложной, и это, бесспорно, нравится не всем. Однако с ней имеют дело лишь *создатели* чарта, а не разработчики приложений. Сложные структуры YAML в шаблоне генерируются из одного-единственного ключа `type`, указанного в файле значений, с помощью которого разработчики изменяют конфигурацию. Файл значений можно указать во время выполнения, что позволяет предоставлять разные файлы (с различными параметрами) для разных кластеров, команд или окружений.

Использование Helm для конфигурации и развертывания как сторонних, так и внутренних приложений, может быть хорошим первым шагом на пути к скрытию какой-то части исходной платформы от разработчиков, чтобы они могли сосредоточиться только на тех параметрах, которые им нужны. Но у этого подхода есть некоторые недостатки. Интерфейс (Values.yaml) по-прежнему имеет формат YAML и может оказаться неудобным, если разработчикам придется анализировать шабло-

ны, чтобы понять последствия вносимых изменений (хотя здесь может помочь хорошая документация).

Если вы не хотите на этом останавливаться, можете создать собственные инструменты, способные представить настраиваемые элементы в виде пользовательского интерфейса. В результате можно задействовать внутри более стандартные средства, а сам интерфейс адаптировать под требования вашей аудитории. Например, вы можете внедрить свои рабочие процессы в существующие средства развертывания (такие как Jenkins) или какую-нибудь систему управления заявками, а внутри в качестве вывода могут по-прежнему использоваться манифесты Kubernetes, которые затем применяются к кластеру. Несмотря на свои богатые возможности, эти модели могут быть сложными в обслуживании, и детали реализации могут в итоге станут известны пользователю.

Интересная реализация этой модели — проект K8s Initializer, недавно выпущенный компанией Ambassador Labs (**https://app.getambassador.io/initializer**). Пользователю предлагается веб-интерфейс, в котором нужно ответить на вопросы о том, какого рода сервис он хочет развернуть, и какая платформа ему нужна. В конце сайт выдает пользователю пакет, который можно скачать и применить к кластеру со всеми видоизмененными параметрами.

У всех методов шаблонизации есть много общих достоинств и недостатков. Мы по-прежнему имеем дело со стандартными объектами Kubernetes, которые применяются к кластеру. Например, генерируя файлы Helm с готовыми значениями, мы оперируем такими объектами, как `Service`, `StatefulSet` и т. д. Это не позволяет нам полностью абстрагировать платформу, поэтому разработчикам все еще нужно *какое-то* понимание внутренностей Kubernetes. Но в этом также заключается *преимущество* данного подхода (при использовании Helm или более абстрактного решения от K8s Initializer). Если готовые чарты Helm или Initializer не генерируют именно то, что вам нужно, у вас остается возможность изменить результаты по своему усмотрению, прежде чем применять их к кластеру.

Kustomize

Kustomize — это гибкий инструмент для применения произвольных операций добавления, удаления и изменения к полям любых YAML-ресурсов Kubernetes. Его можно использовать как отдельно, так и в рамках `kubectl`. Он не занимается шаблонизацией, но может быть полезен при работе с набором манифестов, сгенерированных с помощью Helm, позволяя изменять поля, которые Helm не предоставляет самостоятельно.

На практике по причинам, описанным ранее, Helm служит в качестве средства шаблонизации и подключается к таким инструментам, как Kustomize, для дополнительной гибкости. В результате получается чрезвычайно мощная абстракция, которая дает полную свободу действий. Этот подход находится где-то в середине спектра и зачастую является оптимальным решением для организаций. В следующем разделе мы продолжим двигаться в направлении правого конца спектра абстрагирования и посмотрим, каким образом можно инкапсулировать исходные объекты с

помощью пользовательских ресурсов, специально адаптированных для конкретной организации или отдельного рабочего сценария.

Абстрагирование стандартных компонентов Kubernetes

Как мы уже неоднократно упоминали в этой книге, Kubernetes предоставляет набор элементарных объектов и методов работы с API-интерфейсом. Вместе они позволяют создавать высокоуровневые абстракции и пользовательские ресурсы, описывающие типы и идеи, которые не встроены в платформу. В конце 2019 года в блоге социальной сети Pinterest была опубликована любопытная статья, в которой описывался процесс создания CRD (и сопутствующих контроллеров) для моделирования внутренних приложений. Компания Pinterest применяла этот подход, чтобы скрыть от команд разработки внутренние составные элементы Kubernetes. Обоснование этого метода звучало так: "С одной стороны, стандартной модели сервисов Kubernetes такой, как Deployment, Job и DaemonSet, недостаточно для моделирования наших собственных приложений. Проблемы, связанные с удобством использования, являются огромным камнем преткновения на пути к внедрению Kubernetes. Например, разработчики сервисов жаловались на то, что отсутствие объекта `Ingress` или его некорректная конфигурация нарушали работу их конечных точек. Мы также видели, как пользователи применяли средства шаблонизации, чтобы сгенерировать сотни копий спецификации одного и того же пакетного задания, что превращало отладку в сущий кошмар" (Лида Ли, Джун Лю, Родриго Менезес, Сули Сюй, Гарри Чжан и Роберто Родригез Алькала; "Building a Kubernetes platform at Pinterest", **https://oreil.ly/Ovmgh**).

В листинге 16.3 приведен пример пользовательского ресурса от Pinterest, `PinterestService`. В 25 строках кода уместился процесс создания нескольких стандартных объектов Kubernetes, который занял бы более 350 строк, если создавать эти объекты напрямую.

Листинг 16.3

```
apiVersion: pinterest.com/v1
kind: PinterestService
metadata:
  name: exampleservice
  project: exampleproject
  namespace: default
spec:
  iamrole: role1
  loadbalancer:
    port: 8080
  replicas: 3
  sidecarconfig:
    sidecar1:
```

```
      deps:
      - example.dep
    sidecar2:
      log_level: info
  template:
    spec:
      initcontainers:
      - name: init
        image: gcr.io/kuar-demo/kuard-amd64:1
      containers:
      - name: init
        image: gcr.io/kuar-demo/kuard-amd64:1
```

Это расширенная версия модели шаблонизации, представленной в предыдущем разделе, где конечному пользователю доступны только некоторые параметры. В данном же случае мы можем сформировать входной объект, который имеет смысл в контексте приложения (а не относительно неструктурированного файла Values.yaml) и будет более наглядным для разработчиков. И хотя такой подход не исключает того, что детали реализации просочатся сквозь абстракции, вероятность этого снижается, так как администраторы платформы (создающие CRD/оператор) имеют полный контроль над созданием и изменением исходных ресурсов и могут не ограничиваться имеющимися объектами, как в случае с Helm. Они также могут создавать куда более сложную логику (с помощью контроллера) на языке программирования общего назначения, не ограничиваясь встроенными в Helm функциями.

Однако, как уже обсуждалось ранее, есть и обратная сторона: администраторы платформы должны обладать навыками программирования. Подробнее о создании сервисов и операторов платформы можно почитать в *главе 11*.

Через оператор мы также можем обращаться к внешним API-интерфейсам, чтобы интегрировать дополнительные возможности в наши абстрактные типы объектов. Например, у одного нашего клиента была внутренняя система DNS, в которой должны были регистрироваться все приложения; это позволяло им корректно работать и делало их доступными для внешних клиентов. В рамках сложившейся процедуры разработчики должны были заходить на веб-портал и вручную вводить местоположение своих сервисов, а также порты, которые нужно было пробрасывать от назначенного им доменного имени. Мы можем предложить несколько решений, которые сделают этот процесс более удобным.

Используя встроенные объекты Kubernetes (в данном случае `Ingress`), мы можем создать оператор, который будет считывать специальную аннотацию, указанную в примененном ресурсе `Ingress`, и автоматически регистрировать приложение в DNS-сервисе (листинг 16.4).

Листинг 16.4

```
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
  name: my-app
  annotations:
    company.ingress.required: true
spec:
  rules:
  - host: "my-app"
    http:
      paths:
      - path: /
        backend:
          service:
            name: my-app
            port:
              number: 8000
```

Наш контроллер прочитает аннотацию `company.ingress.required: true` и, в зависимости от названия приложения, пространства имен или каких-то других метаданных, возьмет и зарегистрирует подходящие DNS-записи, а также, возможно, изменит поле `host` с учетом определенных правил. Это избавляет разработчиков от большого объема ручной работы (по созданию записей), но по-прежнему требует создания объектов Kubernetes (в данном случае `Ingress`) и понимания того, что они собой представляют. В этом смысле данный уровень абстрагирования аналогичен тому, который был описан в предыдущем разделе.

Еще один вариант состоит в использовании такого ресурса, как `PinterestService`. В нем инкапсулирована вся нужная нам информация, а с помощью оператора мы можем создать объект `Ingress` и настроить внешние сервисы вроде системы DNS. К разработчикам не просочились никакие внутренние детали, и при этом наша реализация дает нам полную свободу действий.

Поддержка разных уровней абстракции

При выборе подходящего уровня абстракции для своей организации необходимо подумать о том, как будет организован процесс документирования и поддержки. Например, используя что-то вроде Helm (или другое средство шаблонизации) или один из подходов, который сводится к обычным объектам Kubernetes (Pod, PVC, ConfigMap и т.д.), вы получаете в свое распоряжение огромное количество информации, накопленной сообществом, что облегчит диагностику.

Примером этого может быть условие состояния PVC, описывающее, почему не удалось найти хранилище, или, возможно, сообщение об ошибке, которое возвращает Helm при попытке установить чарт с неправильно сконфигурированным шаблоном. Информацию о подобных ситуациях можно легко найти в Интернете благодаря наличию подробной и обширной документации, поскольку участники сообщества делятся своим опытом работы с этими часто встречающимися инструментами и объектами.

Если эти объекты полностью абстрагировать (скрывая их либо за конвейерами, которые не дают прямого доступа к отладочной информации, либо за пользовательскими ресурсами такими, как `PinterestService`), и если область применения этих абстракций очень узкая (возможно, ограниченная лишь вашей организацией), командам разработки будет намного сложнее искать по открытым источникам. В таких условиях хорошая документация незаменима, но при этом на случай экстренных ситуаций необходимо также предоставить резервный доступ к внутренним механизмам.

Резервный доступ бывает чрезвычайно полезным, так как в своей повседневной работе конечные пользователи могут иметь дело с высокоуровневыми абстракциями, предоставляемыми по умолчанию, но при необходимости или желании (в зависимости от индивидуальных навыков и знаний) они могут опуститься на уровень ниже. Как показывает наш опыт, эта модель должна идеально подойти для создания абстракций платформы.

Еще один важный аспект — возможность применения *имеющихся* навыков. Все больше людей приобретают опыт работы с Kubernetes, и им будет легче взаимодействовать с платформой и проводить ее диагностику, если сделать доступными ее внутренности. Кроме того, чем меньше вы видоизменяете платформу (по крайней мере, те ее части, которые доступны пользователям), тем легче вам будет привлечь специалистов, которым не хочется слишком углубляться в уникальные системы/дистрибутивы, чтобы не делать свою специализацию слишком узкой.

Но даже при выборе пользовательских ресурсов и оператора, как обсуждалось в этом разделе, мы все равно открываем для команд разработки кое-какие фундаментальные механизмы платформы. Нам нужно указывать корректные метаданные, версии API-интерфейса и типы ресурсов Kubernetes. Мы также раскрываем файл YAML (если только у нас не предусмотрен конвейер, обертка или пользовательский интерфейс для его создания) и связанные с ним нюансы. В следующем (заключительном) разделе мы переместимся в крайнюю правую часть нашего спектра абстрагирования и поговорим о некоторых технологиях, которые позволяют разработчикам переходить от кода своих приложений сразу к платформе, возможно, даже не догадываясь о существовании Kubernetes.

Полностью скрываем Kubernetes

В предыдущем разделе мы двигались по спектру абстрагирования слева направо, начиная с наименее абстрактных решений (неограниченный доступ к `kubectl`) и теперь заканчивая полностью абстрагированной платформой. Здесь речь пойдет о случаях, когда разработчики даже не догадываются о том, что используют Kubernetes, все их взаимодействие с платформой (более-менее) сводится к фиксации и загрузке кода, что позволяет им сохранять довольно узкую (и углубленную) направленность своей работы и при этом не иметь дела с нюансами платформы.

За последние 10 лет такие провайдеры SaaS, как Heroku и инструменты вроде Cloud Foundry популяризировали модель на основе *загрузки кода*, ориентированную на разработчиков. Суть ее состоит в том, что правильно настроенный инструментарий

предоставляет платформу в качестве услуги (англ. Platform as a Service или PaaS — термин, ставший довольно расплывчатым) со всеми дополнительными компонентами, необходимыми для работы приложения (механизмами наблюдаемости, какими-то средствами маршрутизации/управления трафиком, каталогами программного обеспечения и т. д.), и позволяет разработчикам *просто* загружать код в репозиторий. Специальные компоненты платформы устанавливают подходящие лимиты на ресурсы, создают (при необходимости) среды для выполнения кода и объединяют вместе стандартные инструменты PaaS, чтобы конечным пользователям было удобно с ними работать.

Вы, наверное, уже ожидаете плавный переход к системе Kubernetes, которая тоже поддерживает некоторые механизмы, предоставляющие похожие возможности. Когда платформы PaaS только появились, Docker и Kubernetes не существовали, а применение менее развитых контейнерных приложений было крайне ограниченным. Поэтому данные инструменты изначально создавались для окружений на основе виртуальных машин. Теперь же мы видим, что эти (и новые) инструменты переносят или переписывают с расчетом на Kubernetes как раз по причине, которую мы установили ранее. Kubernetes дает нам очень прочную основу с устоявшимися API-интерфейсами и универсальными механизмами, позволяющими *создавать* эти высокоуровневые платформы.

Систему Kubernetes часто упрекают в том, что она существенно усложняет окружение, как для команды эксплуатации, так и для разработчиков (не говоря уже о фундаментальном переходе на контейнеры, о чем тоже необходимо договариваться). Однако данная точка зрения не учитывает одну из главных целей Kubernetes (которую неоднократно озвучивал один из основателей проекта, Джо Беда): быть платформой *для создания платформ*. Сложные аспекты присутствуют в том или ином виде, но за счет своих архитектурных решений и механизмов Kubernetes позволяет нам переложить эти аспекты на разработчиков и поставщиков платформы, а также на сообщество Open Source, позволяя создавать *поверх* Kubernetes удобные средства разработки и развертывания.

Мы уже упоминали Cloud Foundry — наверное, самую популярную и успешную платформу PaaS (которую в настоящее время переносят на Kubernetes). Существуют и другие довольно зрелые варианты такие, как Google App Engine (наряду с другими бессерверными технологиями) и частично RedHat OpenShift. С развитием этой области мы наблюдаем появление все новых платформ. Одним из таких популярных проектов является Backstage (**https://backstage.io**), изначально созданный компанией Spotify и в настоящее время входящий в CNCF Sandbox. Это платформа для создания порталов, предоставляющая разработчикам специальные абстракции для развертывания и администрирования приложений. И еще до того как мы успели дописать эту главу, компания HashiCorp (разрабатывающая множество открытых облачно-ориентированных инструментов, таких как Vault и Consul) объявила о выходе проекта Waypoint, нового инструмента, который скрывает исходные платформы развертывания от конечных пользователей и предоставляет командам разработки высокоуровневые абстракции. В статье, посвященной этому событию, отмечается следующее: "Мы создали Waypoint по одной простой причине: разработчики

просто хотят развертывать" (Митчелл Хасимото, "Announcing HashiCorp Waypoint", **https://oreil.ly/ZhTJ4**).

Waypoint стремится инкапсулировать этапы сборки, развертывания и выпуска программного обеспечения. Разработчикам по-прежнему нужно создавать (самостоятельно или с использованием вспомогательных средств) конфигурационные файлы вроде Dockerfile, описывающие этот процесс, однако теперь данное описание может охватывать все этапы и при этом быть минимальным, с указанием лишь необходимых параметров. Пример такой конфигурации показан в листинге 16.5.

Листинг 16.5

```
project = "example-nodejs"

app "example-nodejs" {
  labels = {
      "service" = "example-nodejs",
      "env" = "dev"
  }

  build {
    use "pack" {}
    registry {
        use "docker" {
          image = "example-nodejs"
          tag = "1"
          local = true
        }
    }
  }

  deploy {
    use "kubernetes" {
    probe_path = "/"
    }
  }

  release {
    use "kubernetes" {
    }
  }
}
```

Обратите внимание на то, что Waypoint перекладывает на разработчиков *некоторые* обязанности (написание этого файла), однако от них скрывается огромное количество решений. Абстрагирование платформы необязательно означает, что итоговый процесс будет лишен любых сложных аспектов, или что пользователям

больше не нужно учиться ничему новому. Вместо этого, как в данном случае, мы можем предоставить новый, упрощенный интерфейс на *подходящем уровне* абстракции, который обладает оптимальным балансом между скоростью и гибкостью. В случае с Waypoint на этапах развертывания и выпуска можно заменить даже саму исходную платформу и воспользоваться чем-то вроде Nomad от самой компании Hashicorp или какой-то другой системой оркестрации. Платформа абстрагирует все внутренние аспекты и логику. Когда Kubernetes и другие платформы станут более стабильными и *скучными* (кто-то может сказать, что этот момент уже почти наступил), реальные инновации перейдут в плоскость создания высокоуровневых платформ, которые будут помогать командам разработки приносить пользу своим организациям более эффективно.

Резюме

В этой главе мы обсудили разные слои абстракции, которые администраторы платформы могут предложить своим пользователям (обычно группам разработки), а также распространенные инструменты и методики, с помощью которых, по нашему опыту, их реализуют на практике. Если сравнивать с другими областями, любые решения и компромиссы здесь зачастую продиктованы организационной культурой, накопленным опытом, инструментарием, навыками и подобными факторами. Почти все клиенты, с которыми мы работали, решают описанные в этой главе проблемы немного по-своему.

Также необходимо отметить следующее: хоть мы и неоднократно высказывались в пользу того, что команды разработки не должны слишком углубляться во внутренние аспекты платформы развертывания, это вовсе *не* означает, что данный подход всегда верный, или что разработчики никогда не должны знать о том, где и как выполняются их приложения. Например, без этой информации невозможно использовать определенные функции платформы или осуществлять отладку своего программного обеспечения. И, как мы любим добавлять, поиск оптимального баланса зачастую является наиболее удачным путем.

Об авторах

Джош Россо помогает организациям внедрять Kubernetes с момента выхода версии 1.2 (2016 год). За это время он поработал инженером и архитектором в CoreOS (RedHat), Heptio и теперь в VMware. Он участвовал в проектировании и реализации вычислительных платформ для финансовых учреждений, создании граничных вычислительных систем для поддержки 5G и многих других проектах. У него есть опыт работы с окружениями, варьирующимися от аппаратных комплексов, которые администрируются внутри предприятий, до виртуальных машин, предоставляемых облачными провайдерами.

Рич Ландер был в числе первых пользователей Docker и еще в 2015 году начал применять контейнеры для выполнения промышленных рабочих заданий. Он усвоил пользу контейнерной оркестрации на собственном опыте и выполняет промышленные приложения в Kubernetes, начиная с версии 1.3. Используя накопленные знания, Рич впоследствии поработал в командах инженерно-технического обеспечения в таких компаниях, как CoreOS (RedHat), Heptio и VMware, помогая внедрять Kubernetes и облачно-ориентированные технологии предприятиям в производственной, торговой и других сферах.

Александр Бранд начал использовать Kubernetes в 2016 году, помогая в создании одного из первых установщиков этой системы в компании Apprenda. С тех пор Александр поработал в Heptio и VMware, где он занимался проектированием и реализацией платформ на основе Kubernetes для организаций в разных отраслях, включая финансы, здравоохранение, легкую промышленность и др. Являясь программистом по призванию, Александр также участвовал в разработке Kubernetes и других открытых проектов в облачно-ориентированной экосистеме.

Джон Харрис применяет Docker с 2014 года. Он консультирует многие компании из списка Fortune 50, помогая им успешно внедрять контейнерные технологии и методики. Обладая опытом создания облачно-ориентированных архитектур, проектирования и DevOps, он помогает компаниям любых размеров создавать надежные платформы и приложения на основе Kubernetes. Прежде чем перейти в VMware (через Heptio), он работал архитектором в компании Docker, консультируя ее наиболее стратегически важных клиентов.

Об изображении на обложке

На обложке этой книги изображен один из двадцати двух видов китов, которых относят к семейству клюворылых за их рыло, вытянутое в виде клюва, как у дельфинов. О большинстве из этих видов мало что известно ввиду их малочисленности и склонности к обитанию на значительных глубинах вдалеке от континентальных шельфов.

Кювьеров клюворыл (или настоящий клюворыл) — вид, который чаще всего встречается людям. Подобно киту, изображенному на обложке, самцы настоящего клюворыла имеют темно-серый окрас и более светлую голову, самки обычно оранжево-коричневые. Клюворылые киты обладают изогнутым спинным плавником и тонким клювом, что делает их больше похожими на дельфинов, чем на других китов. У самцов вырастают клыки, с помощью которых они отгоняют хищников и иногда сражаются за самок. Возможно, из-за этого именно у самцов кювьерова клюворыла бывают характерные шрамы вдоль боков.

Клюворылые киты добывают себе пропитание, погружаясь на час и более на глубину свыше 490 метров, используя эхолокацию и свой уникальный метод всасывающего кормления, возможный благодаря специальным парам горловых борозд. Ученые высказывают предположения о том, что относительно крупные селезенка и печень клюворылых китов могут быть адаптивным механизмом, помогающим справиться с недостатком кислорода на больших глубинах. За этими глубокими погружениями в поисках пищи обычно следует несколько мелких нырков и продолжительные периоды отдыха на поверхности.

И хотя нам мало что известно о сохранности большинства видов клюворылых китов, четыре из них отнесены организацией МСОП к категории "малого риска, с зависимостью от усилий по сохранению" из-за таких антропогенных факторов, как глубоководное рыболовство, биозагрязнение и гидролокация. Многие животные, изображенные на обложках издательства O'Reilly, находятся под угрозой исчезновения, все они важны для нашей планеты.

Титульная иллюстрация была создана Карен Монтгомери на основе черно-белой гравюры из книги "British Quadrupeds". На обложке используются шрифты Gilroy Semibold и Guardian Sans. Шрифт основного текста — Adobe Minion Pro, шрифт заголовка — Adobe Myriad Condensed, для кода был выбран шрифт Ubuntu Mono от Dalton Maag.

Предметный указатель

E

F

G

H

I

J

K

L

M

N

А

Б

В

Д

Ж

З

И

К

Л

М

О

П

Р



Kubernetes на практике

Несмотря на доминирующее положение платформы Kubernetes на рынке средств оркестрации контейнеров, многие организации, недавно познакомившиеся с этой технологией, испытывают трудности при реализации рабочих проектов. В этом практическом руководстве авторы делятся своим опытом использования Kubernetes в реальных проектах, предлагая решения ключевых проблем.

Гениальность платформы Kubernetes заключается в ее настраиваемости и расширяемости — от поддержки подключаемых сред выполнения до интеграции с хранилищами данных. Эта книга ориентирована на архитекторов платформ, разработчиков ПО, специалистов по информационной безопасности. В ней описан путь к успешному внедрению Kubernetes, который включает в себя освоение различных технологий, методов и практических приемов.

«Столько хороших советов! Жаль, что у меня не было этой книги, когда я только начинал проектировать кластеры».

— **Майкл Гуднесс,** ведущий инженер DevOps, MLB

«Если вам поручили переход на другую платформу или нужно оценить усилия, которые потребуются от инфраструктурных команд для внедрения Kubernetes, эта книга обязательна к прочтению».

— **Даффи Кули,** представитель CNCF

Это руководство поможет вам

- **Оценить возможности создания надежной платформы на основе Kubernetes**
- **Получить практический опыт, который поможет избежать ошибок при создании проектов на основе Kubernetes**
- **Разобраться в том, как архитектура Kubernetes делает возможной разработку расширяемых систем**
- **Взглянуть на вещи глазами внутренних и внешних пользователей и создать платформу, полностью соответствующую их требованиям**
- **Контролировать степень сложности вашей платформы за счет принятия обоснованных решений об абстракциях и используемом инструментарии**
- **Исследовать этапы внедрения Kubernetes, изучить самые современные инструменты, их сильные и слабые стороны**

Джош Россо — инженер с опытом использования Kubernetes в компаниях Red Hat (CoreOS team), Heptio и VMware.

Ричард Ландер — инженер по эксплуатации из VMware, внедряющий на предприятиях Kubernetes и облачно-ориентированные технологии.

Александр Бранд — разработчик программного обеспечения, специализирующийся на Kubernetes и облачно-ориентированных технологиях.

Джон Харрис — ведущий инженер с опытом работы над облачно-ориентированными инструментами, платформами и методиками в компаниях VMware, Heptio и Docker.



191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

